Report No. UIUCDCS-R-79-972 UILU-ENG 79 1719

AUTOMATED DESIGN OF DIGITAL MULTIPLEXERS

by

Albert Ernest Casavant

June 1979 NSF-OCA-MCS77-27910-000041

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-79-972

AUTOMATED DESIGN OF DIGITAL MULTIPLEXERS*

by

Albert Ernest Casavant

June 1979

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF MCS77-27910 and was submitted in partial fulfillment of the requirements for the Master of Science in Computer Science, June 1979.

ACKNOWLEDGEMENT

I would like to thank my advisors, Professor Dan Gajski and Professor David Kuck, for their help, advice, and the computing resources they made available to me during the rocky road leading to completion of this thesis.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

Page

Page

CHAPTER 1

Algorithmic Description

1.1 Introduction

In recent years design automation has become an increasingly important part of the digital system design process. It is now widely used for IC layout and artwork generation [1, 2] and for the PC board layout and interconnection [3]. These examples can properly be called physical design automation, since they are concerned primarily with the physical aspects and not the logical aspects of digital system design. Also common now are logic simulation programs which have come to be included under the catchall phrase "design automation."

With the development of VLSI technology, many semiconductor and computer manufacturers are faced with design complexities which made traditional logic design methods inefficient and in extreme cases impossible. So it seems natural to attempt to extend the scope of design automation to include logic design and systems design. Unfortunately, many unanswered questions plague those who attempt this extension, not the least of which is how to represent a digital design at a high level. Also, no efficient algorithms to perform automated logic design exist which are able to produce designs that approach the quality of those produced by an experienced human logic designer.

The representation problem has received the most attention recently, as evidenced by a plethora of logic design languages (e.g., [4, 5, 6, 7]. Work at Stanford has concentrated on the structural aspects of logic

design representation [8] and the LOGOS system [9] has contributed substantially to architectural level representation and simulation.

The algorithmic problem has received much less attention. Notable papers in this area include [10, 11, 12]. None of these approaches attempt to go above the gate level. The largest component dealt with is a group of gates assembled into a package rather than with entire adders, multiplexers, shift registers, etc. The most advanced attempt at hardware synthesis to date is the Carnegie-Mellon system [13, 14]. This system performs some optimization on designs at the register transfer level. A disadvantage, however, is that the user must be conscious of the level of parallelism he desires, the control unit design, and detailed timing and component delays. This paper details the methods used to design one common component of digital systems using software tools which assume the burden of determining component delay and package assignment. A companion paper [15] describes a graphical display and editing system having general application as well as specific utilities dedicated to the multiplexer problem.

A multiplexer (MUX) is a basic component of many digital circuits and is used in such applications as data selectors, parallel to serial conversion and Boolean function generation. In this paper the design automation of multiplexers consists of translating a simple non-technical description of multiplexer tree(s) into a hardware implementation using building block components from a predefined library of components. The translation process is a pseudo-optimization process in which the user inputs the multiplexer design automation tool (MDAT) with target

values of critical parameters and weights specifying the relative importance of meeting the target values in the final design. The library of components consists of various component types (CTs) e.g. a 2 line to 1 line (2:1) MUX, 4 line to 1 line (4:1) MUX, etc. and selections of components of each type, henceforth called the indexed components of that type. Indexed components correspond to differences in circuit implementation of a type, e.g. number of logic levels or a particular logic technology such as TTL, LP-TTL, S-TTL, LS-TTL, ECL, etc. Please note that the multiplexers described herein are not those which have so-called "3-state" output. Thus wire-or'ing of outputs is not permitted. The MDAT translation process described here in over simplified form consists of applying heuristics to obtain a covering of multiplexer trees obtained from the input specification using the largest possible CTs while taking advantage of commonality of components among trees, and then substituting indexed components for each type to maximize the merit factor, a measure of the closeness of the overall design to the target values of critical parameters. If the MDAT is added to other design automation tools for other basic components of digital systems and augmented with a graphics capability for added versatility, a complete interactive design automation system may be constructed.

1.2 Input description and interpretation

1.2.1 Catalog

The catalog is the predefined library of components from which the hardware implementation of multiplexers is constructed. Each entry in the catalog defines a component type (CT), gives a package constraint for the type, and lists the indexed components associated with the type. The entries are implicitly ordered by increasing size of CTs e.g. 2:1 MUX followed by 4:1 MUX, etc. Missing sizes in the sequence must be given null entries in the catalog. The package constraint is an integer describing the number of components of this type residing in one integrated circuit package; a zero for this integer indicates a null entry. Each indexed component within the list associated with each entry is described by three integers: cost, power, and delay - each expressed in a unit which is consistent throughout the entire input description. Convenient units for these quantities are cents, milliwatts, and nanoseconds, respectively. Note that these three quantities are not the totals for an integrated circuit package but rather for each indexed component within the IC. A BNF description of the catalog format is given in fig. 38 of 3.2.

1.2.2 MUX design input information

Definition 1: Each MUX inputted to the MDAT has a set of inputs, I_j, $1 <= j <= M$ where M is the number of MUX trees given in the input information and j is the jth of these trees.

In preparation for the following discussion consider the following:

C(S) is the cardinality of set S.

N is max $C(I_j)$, $1 <= j <= M$.

Z_k is $\{1,\ldots,k\}$.

Q is the number of inputs to the largest size CT inputted to the MDAT.

$P(Z_k)$ is the set of all setsets (power set) of Z_K.

Definition 2: An output vector V_l is $\langle i_{l_1}, i_{l_2}, \ldots, i_{l_M} \rangle$ where $i_{l_j} \in I_j \cup \{d\} \cup \{x\}$ (d is don't care, x is dummy input) and $1 <= l <= N$.

Definition 3: A multiple output mux (MOMUX) is an m + 1 tuple, $(I_1, I_2, \ldots, I_M; S)$, where $S \subseteq \{V_l\}$, $1 <= l <= N$.

Definition 4: The input select code is a binary integer which uniquely identifies each vector V_ℓ.

Assumption 1: The input select code which selects a certain input to be the output of the multiplexer is determined by the MDAT, not the user. The only restriction on this choice is in the case of multiple output multiplexers; the code selected must preserve the output vectors specified by the user. For example, if the user specifies that an arbitrary select code is to place inputs 5, 8, and 12 of multiplexers 1, 2, and 3 on their respective outputs then the code selected by the MDAT must guarantee that the same select code, whatever it may finally be, applied simultaneously to multiplexers 1,2, and 3 must place inputs 5, 8, and 12 on the outputs.

Assumption 2: The output vectors of MOMUXs are specified in arbitrary sequence such that if there is a don't care entry for a particular output, no specified entries for that output may appear later in the sequence. This restriction is desirable to generate binary trees with a minimum of nodes. Each node is a 2:1 MUX. It may be necessary for the user to reorder trees and reorder inputs to each tree to satisfy this assumption. Dummy inputs may also be required. An

overall objective should be to minimize tree nodes. Fig. 1 demonstrates the restrictions imposed by this assumption.

Associated with each input is a delay which is the length of time between the last clock pulse in a sequential circuit (or some other fixed time base) and the arrival of a signal at the input to the multiplexer. This allows the MDAT to select hardware components based on this information and may result in less expensive implementations. Inputs to the MOMUXs which represent the same physical input should have the same integer value.

Three other packets of data are the MOMUX delays, targets, and weights. The latter two are discussed in 1.2.3. The total time measured from the last clock pulse (or some other fixed time base) until the appearance of a valid signal on a particular output of a MOMUX is defined as the tree delay .i.e. the target delay for the entire MUX tree. A BNF description of the MUX design input information is given in fig. 36 of 3.2.

1.2.3 Merit factor description

****

tree

1	2	3
A	I	N
B	J	O
C	d	d
D	d	d
E	K	P
F	d	Q
G	M	d
H	d	d

Violation of assumption 2 by rows E..K..P and F..d..Q.
(d = don't care)

tree

1	2	3
A	I	N
B	J	O
E	K	M
F	d	Q
G	M	d
C	d	d
D	d	d
H	d	d

Rows have been rearranged but row F..d..Q still a problem.

tree

1	2	3
A	I	N
B	J	O
E	K	P
F	x	Q
G	M	d
C	d	d
D	d	d
H	d	d

Assumption 2 is now satisfied using dummy input x in row F..x..Q.

fig. 1: Example of assumption 2

****

As mentioned above, the merit factor plays a key role in the optimization process. The merit factor is mainly for the internal use of the MDAT and when it is outputted it is multiplied by 100. The merit factor always lies between +1 and -3 and is calculated as shown in fig. 2. TARCOST, TARPOW, and TARPACKS are the target values for cost, power, and number of integrated circuit packages respectively for the entire MOMUX. The over-weights WTCOSTO, WTPOWO, and WTPCKSO (all between 0 and 10^y where y is chosen to give an adequate resolution) give the relative desirability of exceeding the targets for cost, power, and number of IC packages, respectively. As an example for the case WTPOWO > WTCOSTO, if the MDAT has chosen two possible hardware implementations for a MOMUX having MPOW1 = TARPOW, TARCOST/MCOST1 = .8 and TARPOW/MPOW2 = .8, MCOST2 = TARCOST. The first implementation is considered more desirable. Note that in the case of over-weights, if one critical parameter such as power is given a higher weight than cost, exceeding the target cost-wise is more desirable than exceeding it powerwise. The reverse relationship is true for under-weights. In setting the range for the merit factor it is assumed than at least one under-weight is equal to 100. The under-weights are designated WTCOSTU, WTPOWU, and WTPCKSU.

```
**************************************************************

MF        = merit factor
POF       = power factor
COF       = cost factor
PAF       = pack factor
N         = normalization


N = WTCOST + WTPOW + WTPCKSU


MF = (COF + POF + PAF)/N


(Leave out PAF and WTPCKSU for optimization without  package
constraint)


if MCOST > TARCOST
then COF = -((MPOW - TARCOST)/MCOST) * WTCOSTO
else COF = -((TARCOST - MCOST)/TARCOST) * WTCOSTU


if MPOW > TARPOW
then COF = -((MPOW = TARPOW)/MPOW) * WTPOWO
else COF = ((TARPOW = MPOW)/TARPOW) * WTPOWU


if MPACK > TARPACKS
then PAF = -((MPACK - TARPACKS/MPACK) * WTPCKSO
else PAF = ((TARPACKS - MPACK)/TARPACKS) * WTPCKSU
```

fig. 2: Merit factor calculations

```
**************************************************************
```

1.3 Commonality and covering

The first step in determining commonality which can be applied to an implementation is to detect all commonality candidates (CCs) and assign a priority to each.

Definition 5: A commonality candidate of size i (CCi) is the set $\bigcap_S I_j$, $S \in P(Z_M)$ and $C(S) = i$.

A commonality candidate is any CCi. To indicate the relative desirability of using a given CC, CCs are given priorities and entered into a list.

Definition 6: A priority level (PL_{ij}) is $\{CCi \mid C(CCi) = j\}$, $2 <= i <= Q$ and $1 <= j <= M$.

Assumption 3: The ordering of priority levels is as follows: $P_{ij} > P_{kl}$ iff $i = k$ and $j >= l$, or $i > k$, where $(2 <= i,k <= N)$ and $(2 <= j,l <= M)$.

There can be any number of entries in each priority level, limited of course by the overall size of the problem. Within each priority level there is no assigned ordering; the ordering is a function of the algorithm for generation of all commonality candidates. Only the largest commons (in

the sense of largest input sets common among a set of MOMUX trees) appear in the list initially.

Definition 7: The input relation for inputs p and q is defined as follows:

p >= q iff

p is not d where d is don't care

and q may be any input including d and x

(the dummy input)

Definition 8: A vector relation is defined as follows:

$V_k >= V_l$ iff

$i_{kr} >= i_{lr}$ for $1 <= r <= M$

Definition 9: A gapless select code is a sequence of binary integers corresponding to and ordered sequence of output vectors such that $V_k >= V_l$ for every pair of vectors in the sequence.

Not all of the CCs may be used in the hardware implementation. Assumption 2 dictates that to build binary trees no gaps may appear in the input select code. This implies that a CC among two or more trees of a MOMUX must possess a gapless select code sequence of output vectors corresponding to the sequence of inputs in the CC. Nonvalid and valid common candidates (VCCs) are shown in fig. 3.

****

```
      tree
1 .. 2 .. 3
---------------
A .. Q .. K
B .. R .. L
C .. S .. M
D .. S .. N
E .. A .. O
F .. C .. P
G .. B
H .. D
I
J
```

Common grouping ABCD would produce a gap in the select code if it were admitted as a VCC.

```
      tree
1 .. 2 .. 3
-------------
A .. Q .. K
B .. R .. L
C .. S .. M
D .. T .. N
E .. A .. O
F .. B .. P
G .. C
H .. D
I
J
```

In this case ABCD is a valid common grouping.

(Cap letters represent inputs.)

fig. 3: Valid and nonvalid common groupings

****

Another less obvious restriction on the use of CCs is that the acceptance of one VCC may invalidate another CC by causing the second CC to violate assumption 2 (see fig. 4). In the following discussion it will be helpful to think of the inputs to the MDAT as columns of inputs, each column representing the inputs which form a certain output of the MOMUX. This representation has already been introduced in figs. 1, 3, and 4.

> Definition 10: The row position (RP) is an integer giving the natural ordering of rows in the column/row representation of the input data to the MDAT ($1 <= RP <= N$).

> Definition 11: A frame $F_{i,j}$ for CT size i is $\{RP \mid (j * (i - 1)) + 1 <= RP <= j * i$ where $1 <=_{\wedge}^{i<=} Q$ and $1 <=_{\wedge}^{j<=} \lfloor N/Q \rfloor$ and j is some integer.

Due to the complexity of checking for VCCs and to increase efficiency by eliminating as many possibilities with as little computation as possible, the commonality checking process is done in stages. Very few CCs on the average will reach the last and most computationally complex stages of the process. The commonality search procedure (CSP) is conducted as follows:

****

```
      tree
1 .. 2 .. 3
-------------
A .. F .. K
B .. I .. L
C .. H .. M
D .. H .. M
E .. B .. P
F .. A .. O
E .. B .. P
H .. C
G .. D
```

Assuming that ABCD has been accepted as a VCC, attempting to make FH a common would violate assumption 2 since a modified configuration would look as follows:

```
      tree
1 .. 2 .. 3
-------------
A .. F .. K
C .. H .. M
B .. I .. L
D .. J .. N
F .. A .. R
H .. C
E .. B .. P
G .. D
```

Now there is a gap in the select code

fig. 4: Common invalidation

****

Assuming that the procedure has worked its way down (in the direction of highest to lowest priority level) to a certain size (i) of CC covering a certain number of columns (j) , i.e. to priority level P^{L}_{ij}, possible placements of the CC on the columns of inputs are checked until a VCC is found or all placements are exhausted. Valid placement for a CC of PL_{ij} in column l is frame F_{ik} where $1 <= k <= \lfloor C(I_\ell)/C(CC) \rfloor$. If a VCC is not found and the number of columns covered by the CC is greater than two then the number of columns to be covered is reduced by one. This CC is then assigned a new priority among all CCs which reflects its reduced column choice size. If the number of columns is two, the CC size, if it is greater than two, is reduced by one; otherwise the CSP proceeds to the highest priority level having CC size one less than the present CC size. If the CC size is two and the column choice size is two, the CSP is at the lowest priority level and when the CCs at this priority level are exhausted the CSP terminates.

Returning to the determination of VCCs, the first screening process begins with the determination for each input in the CC its range of row position movement (RRPM).

Definition 12: The RRPM for an input is the range of RPs the input may occupy subject to the restrictions of assumption 2.

The row position movement is restricted by the presence of inputs within VCCs at the same RP as the input of the CC under test. Since we have assigned a priority to all CCs, the VCC already determined has precedence over any VCC candidate. If any inputs in a VCC are moved to RPs outside of the RRPM of the VCC then this VCC would become invalid (see fig. 4). Once the RRPM of all inputs of the CC has been determined, the set union of the RRPMs of each input in the CC are intersected with the valid placements referred to above in the CSP to determine if a gapless select code can be constructed which meets the RRPM for each input and simultaneously has a valid placement in a particular column (test 1).

Test 1 is illustrated by fig. 5. A CC passing test 1 may still violate assumption 2 and more checking is required.

Definition 13: The number of columns (NC) at a particular RP is: $\sum_{1 \le j \le M} r_j$ where $r_j = 1$ iff $i_{RP,j}$ for a vector occupying this RP is not d, otherwise $r_j = \emptyset$.

****

```
      tree
1 .. 2 .. 3
------------
A .. E .. P
B .. F .. B
C .. G .. C
D .. H .. R
E .. I .. S
K .. J .. T
L .. A .. U
F .. B .. V
M .. C .. M
N .. D .. N
```

If ABCD and MN are VCCs, then an attempt to make BC in columns 1,2 and 3 a common would violate test 1 since C in column 2 cannot assume an RP which would give BC a valid placement.

fig. 5: Test 1

****

Definition 14: A set of NCs (SNC) is the set of NCs corresponding to the RP of each input of a CC in a particular column.

Clearly, to avoid gaps in the select code, the SNCs for the RPs of the inputs in the CCi covering j columns must correspond to the SNCs of RPs in the frames for the CC (test 2). Furthermore, since the inputs in each VCC must be able to be put in some order in every column involved, order checking must be done (test 3). Fig. 6 demonstrates a case where this is necessary.

Up to this point in the determination of VCCs, the effects of VCCs already placed has been ignored. The grouping and ordering necessary for a VCC candidate may affect the grouping and ordering of VCCs whose inputs have the same RPs as the inputs of the VCC candidate in the columns involved. Figs. 7 and 8 demonstrates two cases of this problem. Test 4 and test 5 check for these cases.

When the CSP ends, the inputs within VCCs must be ordered. This was implicitly done in the tests above but for the benefit of outputting the results of the CSP, a consistent explicit ordering is assigned. Although not mentioned above, the CSP serves as a covering algorithm for the inputs in VCCs since the sizes of CCs are in one to one

****

```
      tree
1 .. 2 .. 3
-------------
E .. A .. K
F .. B .. L
G .. C
H .. D
A .. I
C .. J
B
D
```

Test 2 has clearly been passed here for CC ABCD since the set of inputs A,B,C, and D may assume some order (possibly different in each column) which gives an SNC equal to 3,3,2,2 for column 2 inputs and 2,2,1,1 for column 1 inputs; however, any ordering of ABCD which is identical in columns 1 and 2 will cause a violation of assumption 2 such as:

```
      tree
1 .. 2 .. 3
-------------
E .. A .. K
F .. B .. L
G .. C
H .. D
A .. I
B
C .. J
D
```

fig. 6: Test 3

****

****

```
      columns
P .. Q .. I .. J
-----------------
G .. A .. K .. E
H .. B .. L .. Q
I .. C .. M .. R
J .. D .. N .. F
K .. O .. E .. S
L .. P .. F .. T
A .. U .. V
B .. W
C .. X
D
```

Assume ABCD is a VCC; now an attempt to make EF a VCC violates test 4 since EF grouped together in columns I and J implies AD grouped together in columns P and Q which violates assumption 2 when this grouping (imposed by grouping EF) is assumed for AD of ABCD in column P.

fig. 7: Test 4

****

****

```
      column
1 .. 2 .. 3 .. 4 .. 5
----------------------
A .. x .. R .. x .. E
B .. x .. x .. x .. F
C .. x .. S .. x .. G
D .. x .. x .. x .. H
R .. x .. x .. E
x .. x .. x .. F
S .. x .. x .. G
x .. x .. x .. H
x .. A .. x
x .. B .. x
x .. C
x .. D
```

x = dummy input

Assume that ABCD and EFGH are VCCs. Now test RS for commonality. RS causes inputs E and G to be grouped together which in turn cause A and C to be grouped together, but this grouping of A and C violates assumption 2 in column 2.

fig. 8: Test 5

****

correspondence with the sizes of CTs.

> Definition 15: A covering algorithm (in the sense the term is used in this paper) consists of covering the nodes of binary trees with CTs which are binary trees themselves i.e. a 4:1 CT is a binary tree of 3 nodes, an 8:1 CT is a binary tree of 7 nodes, etc.

To complete the covering, inputs not in VCCs are arbitrarily split into the largest groups possible, where each group size corresponds to a CT size.

It is useful here to stop and look at other assumptions which have been made thus far.

> Assumption 4: All commonality possible is exploited in the CSP although doing this may not produce an optimal hardware implementation.

Possible non-optimal effects fall into two categories. In some cases large non-common CTs may be more desirable than a mixture of small common CTs and non-common CTs. Also using all commonality has the effect of fragmenting large common CTs into smaller ones. The effect of using all commonality possible cannot be determined exactly until the optimization phase. Since bringing the CSP into the

optimization loop would involve the examination of many more possibilities than is presently done, a decision was made to arbitrarily use assumption 4. Since the use of commonality reduces the number of nodes in the MOMUX tree, the positive effect of higher integration achieved by using large non-common CTs would have to be strong enough to overcome a decrease in numbers of nodes to be covered to make assumption 4 unreasonable.

Assumption 5: All inputs are given equal weight for grouping purposes.

Since the inputs have delays associated with them (see 1.2.2), the grouping of inputs could significantly affect the choice of indexed components within particular CTs. Hence in a true optimization, optional groupings such as may occur in the choice of coverings for inputs not in VCCs should be taken into account.

Assumption 6: When commonality is not involved, the largest possible groupings of inputs are chosen.

This may seem like an obvious assumption but there are cases when the largest groupings are not the best, such as a very high cost penalty for a high level of integration.

1.4 Optimization

After converting the column and row data structure to a tree-type data structure we are almost ready to select indexed components for each CT represented in the data structure. To increase the efficiency of the optimization process the MOMUX tree data structure is partitioned into independently analyzable blocks (IABs). Each IAB may undergo optimization independently since it shares no CTs with other IABs (see fig. 9).

The four critical parameters of a MOMUX are cost, power, delay, and number of IC packages. The package constraint aspects of optimization will be considered in 1.5. To restrict the possibilities to be checked, one of the remaining parameters is singled out for special treatment.

Assumption 7: The target for tree delay inputted to the MDAT will not be exceeded.

This greatly limits the choices for indexed components as well as limiting the number of components which must be considered for each path. Another data structure is introduced which corresponds to paths in the MOMUX. These

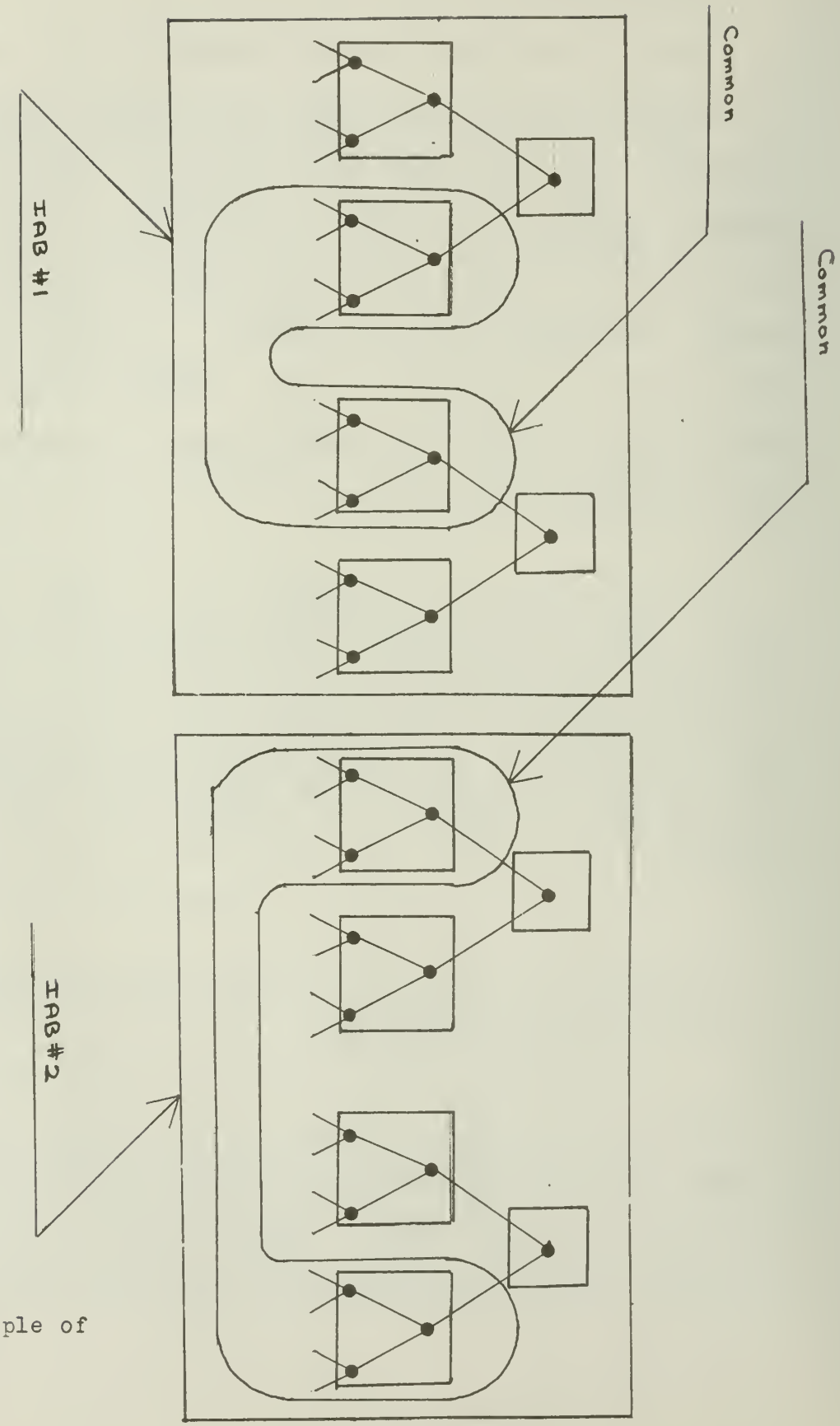


fig 9: Example of IABs

are not the same as paths in a graph. In fig. 10 an IAB with 6 paths is shown. Paths are selected as follows:

> Starting at the root of the leftmost tree of the MOMUX, traverse the tree by always taking the leftmost branch leaving a CT unless the CT to the left is part of a VCC or no CT exists to the left. If no CT exists to the left, a path terminates. If a common CT is to the left, then traverse the tree by taking the next CT to the right of the common which is not also common. The CT chosen (if there is one) becomes the beginning of a new path. When all of the ancestors of a CT have been thus examined, return to the predecessor of this CT and continue in the same fashion. When the first tree in the IAB is exhausted, move on to the next (if there is one). When all CTs involved in a VCC have been examined, this collection of (at least two) CTs becomes a path.

Note that when paths are chosen and ordered in the way described, a choice of indexed components in all previous paths in the list is sufficient to describe the delay constraints for the path under consideration (except in certain cases described below). For example, when indexed components are chosen for paths 1, 2, 3, and 4 in fig. 10 the delay constraint for path 5 is fixed as follows:

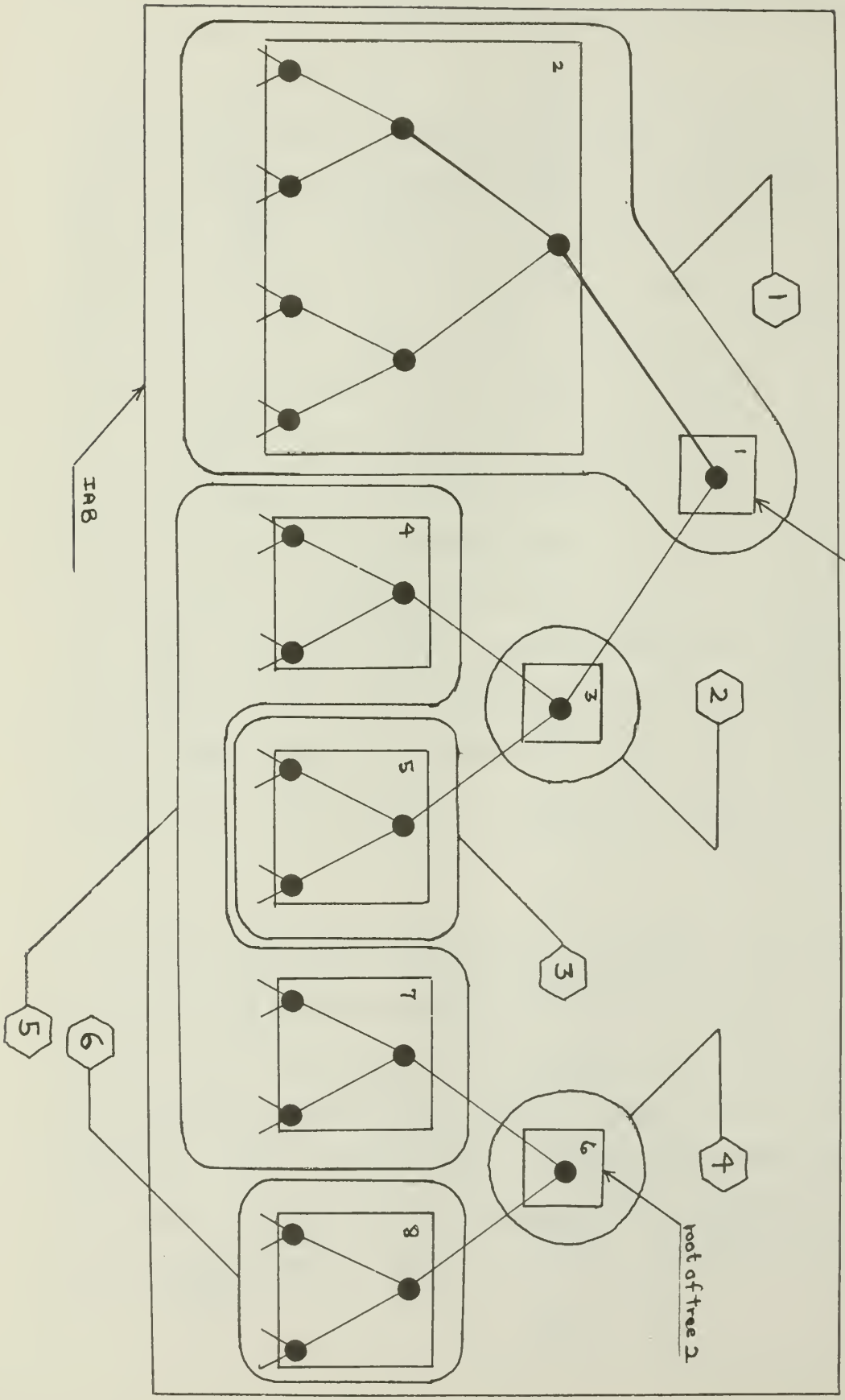


fig. 10:
Example of
paths

Assume that the delay at the root of tree 1 is fixed at the maximum allowed i.e. the target. The maximum allowable delay for indexed component 4 is then

$$D_{COMP4} = D_{TREE1} - D_{COMP1} - D_{COMP3} - D_{MAX4}$$

where D_{MAX4} is the maximum of all input delays to component 4. Similarly the delay for indexed component 7 is

$$D_{TREE2} - D_{COMP6} - D_{MAX7}$$

Note that when a path begins at the tree root or does not terminate at inputs (called a special path, SP), an arbitrary choice of indexed components is permitted. Examples are paths 1, 2, and 4 in fig. 10.

The optimization procedure (OP) proceeds as follows:

For each IAB, indexed components are assigned to paths such that an arbitrary choice is made for SPs not beginning at the tree root and a choice is made based on delay constraints external to the path in the case of SPs beginning at a tree root or non-SPs. When an assignment cannot be made because of a delay constraint violation, control returns to the last SP in the sequence of paths prior to the path in violation. When

a new assignment is made for the SP, the forward movement through the sequence of paths continues again. Two possible final outcomes are: 1. no assignment of indexed components is possible which meets delay constraints, or 2. an assignment is found meeting delay constraints. In the former case all optimization activities cease and an error message is issued indicating which IAB was involved. In the latter case the merit factor for this IAB is calculated without package constraint (see 1.2.3). If the calculated merit factor is higher than any calculated so far for the IAB involved then the indexed component assignments are stored and the OP is reentered to look for an indexed component assignment yielding a higher merit factor. The OP is repeated for each IAB of the MOMUX.

1.5 Packaging Considerations

So far in the interests of simplicity and efficiency, the IC packages housing indexed components have been ignored. Consider, for example, the fact that if IC packages were included in the OP above, it would no longer be possible to form IABs. This is another case of a phenomenon which occurs often in logic design where the logical boundaries of circuit entities do not correspond to

physical boundaries. Although assigning packages after one pass through the OP is possible it would probably not produce a solution as good as one produced by a competent logic designer. Note that each level of the MOMUX trees when viewed as binary trees composed of 2:1 CTs corresponds to one bit of the select code. Thus a 4:1 CT at levels 5 and 6 of a MOMUX tree and a 4:1 CT at levels 3 and 4 cannot be placed in the same IC package having two 4:1 CTs, since each package is assumed to have only one set of select inputs (usually the case for practical ICs). Thus, after one pass through the OP, we could categorize the indexed components at each level of the tree by CT and indexed component within each CT. The procedure is best illustrated by an example.

Suppose at a given level there are 8 indexed components of CT 2:1. Their indices and delays (I,D) are as follows:

(1:5Ø)(1:5Ø)(2:25)(2:25)(2:25)(3:1Ø)(3:1Ø)(4:5)

and 4 CT 2:1s are in each IC package. A solution which preserves the delay target is to assign these 8 indexed components to two CT 2:1 packages composed of four components each of indices 2 and 4 so that the final assignment is:

(2:25)(2:25)(2:25)(2:25)(4:5)(4:5)(4:5)(4:5)

This method may be used at each level of the MOMUX trees and a solution obtained. Since the overall speed of the MOMUX is now probably faster than necessary we end up with a solution costing more and requiring more power than is absolutely necessary.

An alternate approach is to assign one level at a time to packages starting at the roots of trees and then reenter OP so the the OP may take advantage of possibly faster and now fixed assignments at upper levels of MOMUX trees. The OP now may alter previous selections at lower levels by using slower indexed components. After the IABs have been optimized as in 1.4, another level of the tree is fixed by assigning packages. This continues until all levels in the tree are fixed and a final solution achieved.

1.6 Summary of Algorithm

The algorithm is summarized in figs. 11, 12, and 13.

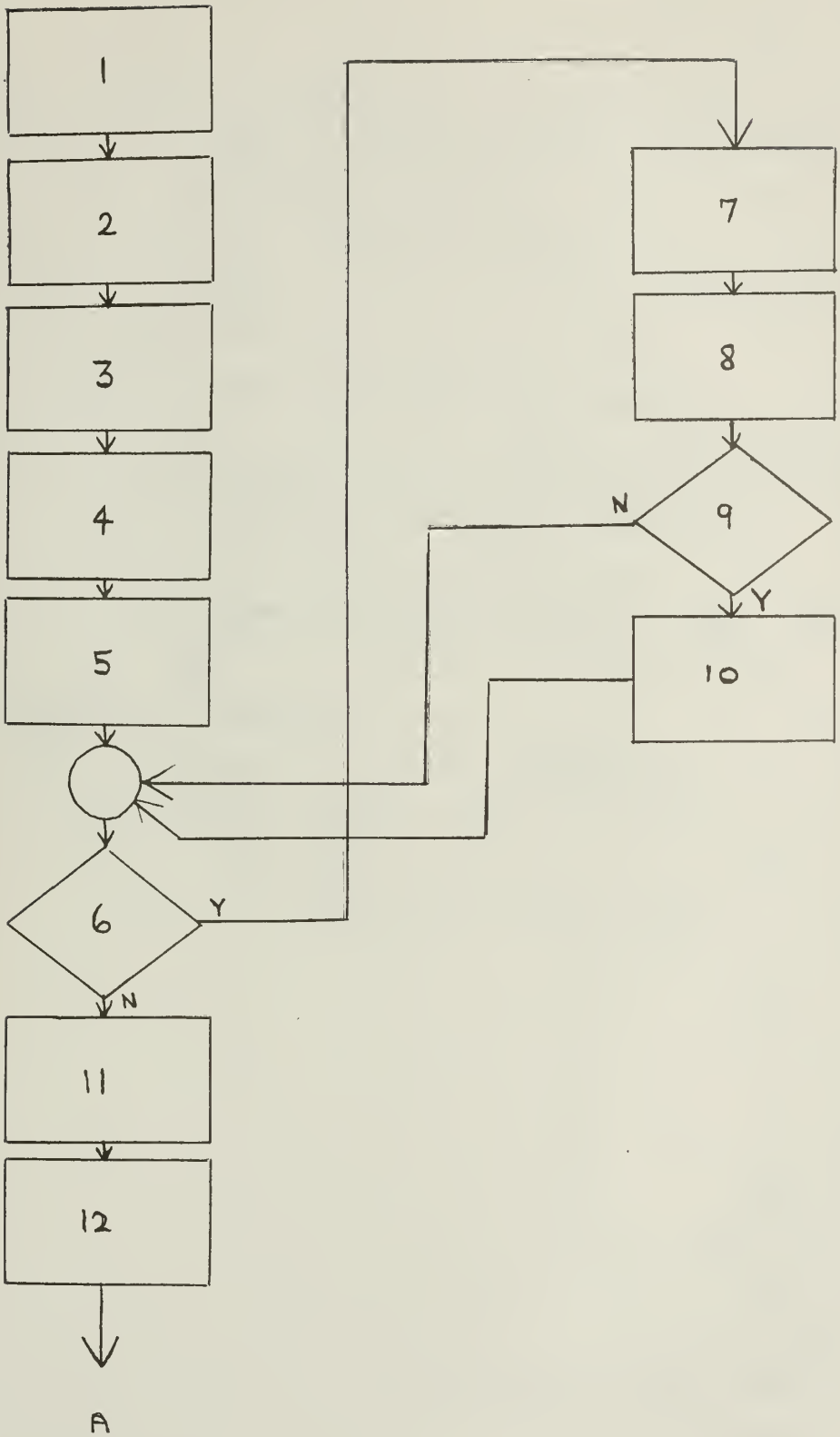


fig. 11: Summary of algorithm

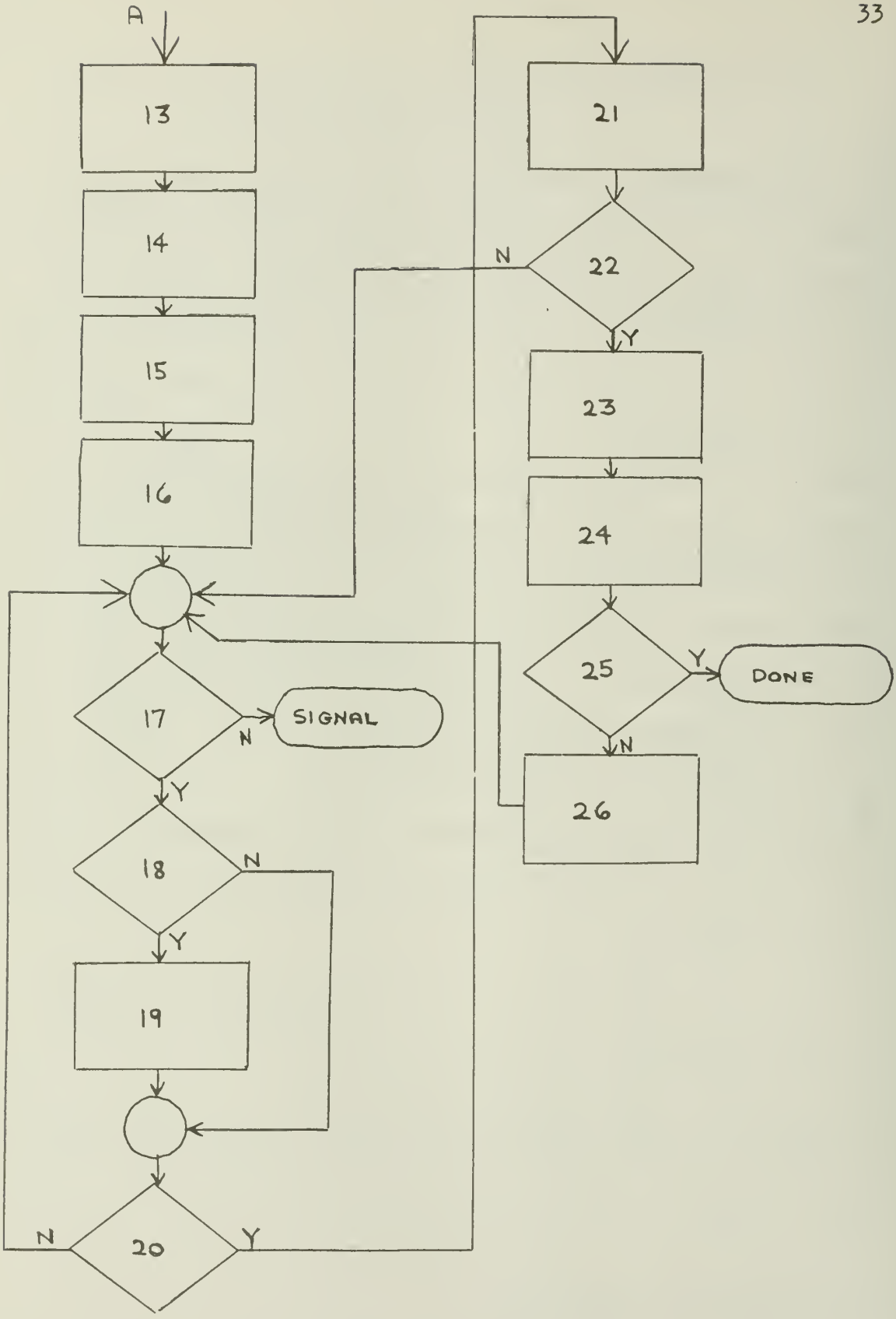


fig. 12: Summary of algorithm (continued)

1. read in catalog
2. read in design information
3. construct column and row data structure (DS1)
4. generate all common candidates (CCs)
5. construct CC data structure (DS2)
6. are there more entries in DS2?
7. select next CC
8. perform tests 1 - 5 to check for VCC
9. is this CC a VCC?
10. modify DS2 to reflect new VCC
11. arrange VCCs
12. complete covering of non-VCCs
13. generate tree data structure DS3 using DS1
14. generate IABs
15. generate path data structures DS4A, DS4B and DS4C
16. initialize fix level, IAB count, and merit factor
17. can find an assignment of indexed components for this IAB?
18. merit factor for this IAB best so far?
19. modify DS3B to show new assignment
20. exhausted all combinations of indexed components for this IAB?
21. increment IAB count
22. exhausted all IABs?
23. create package data structure (DS5)
24. assign packages
25. is fixed level the bottom level of tree?
26. increment fixed level, increment IAB count and MF

fig. 13: Algorithm flowchart key

CHAPTER 2

Program Description

2.1 Overall program structure

The correspondence between procedures and entities in the flowchart (figs. 11, 12, and 13) is given in figs. 14 and 15.

2.2 Data structures

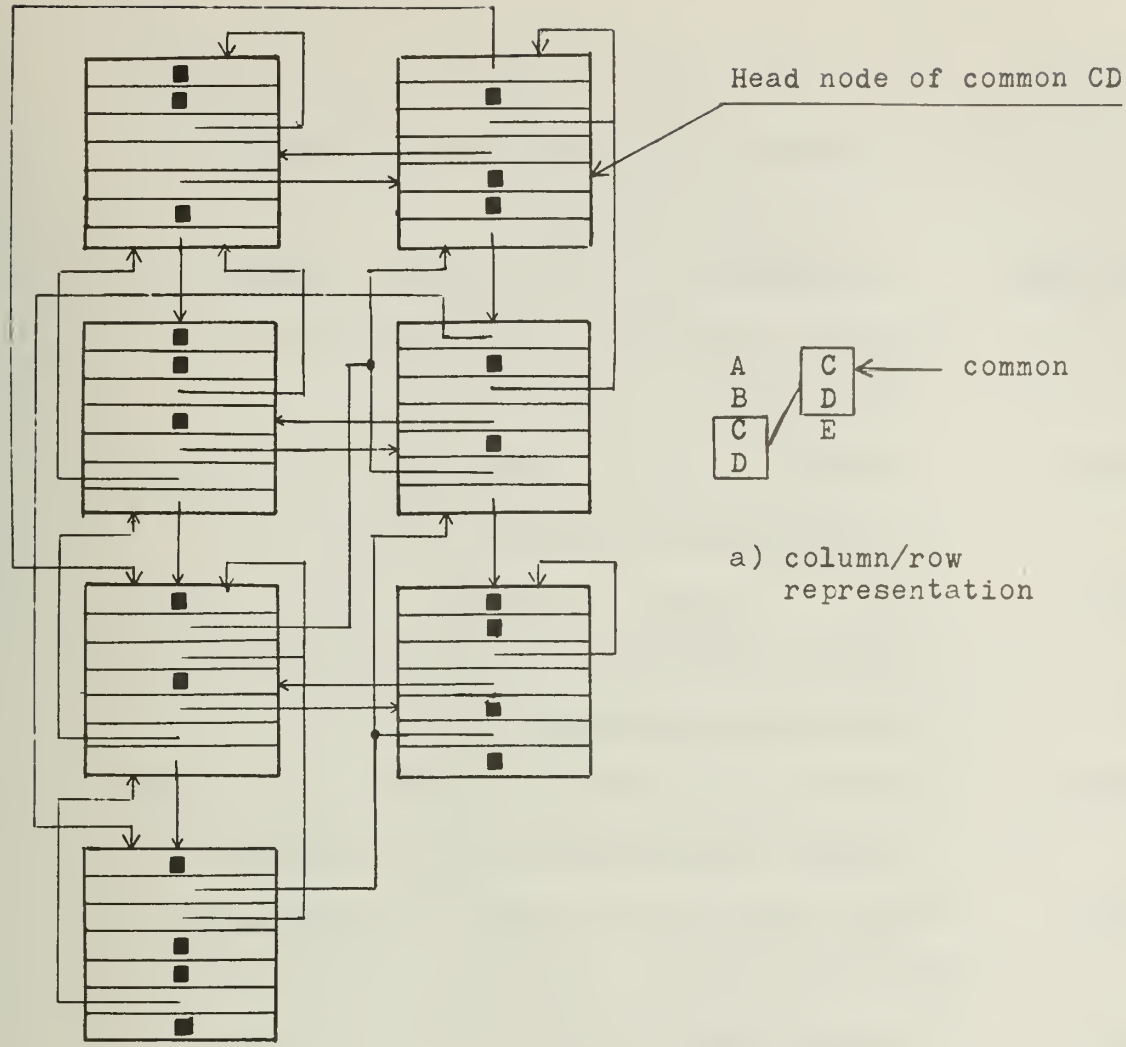
The pointers of DS1 (column/row data structure) are shown in fig. 16. The remainder of record BLOCKK is as follows:

Procedure	Flowchart code
READINFO	1,2
DELCOM	utiity
CREATECOM	5
GETCOMB	utility
GENSET	4
MOVELE	utility
INTERSECT	6,7,8,9,10
GENLVLS	utility
INIT	utility
FINDSETS	8
OKCHK	8
MOVEM	utility
CHKLVLS	8
CHKPLC	8
PLC	8
ARRANGE	11
BRKUP	12

fig. 14: Program structure

procedure	flowchart code
GENCOL	3
MAKETREE	13
NEWNODE	utility
PERTREE	utility
LOGOF	utility
PACKS	13
SUPERPACK	utility
MARK	utility
FINDELE	utility
FINDMAX	utility
RELATIONAL	14
INCL	utility
MERIT	utility
FILLINFO	utility
CNEWPATH	utility
HANDLECOM	15
CPATHNONCOM	15
OPTIM	17
INCIND	utility
MOVPTR	utility
CHGTYPE	17
CALCDLYHI	utility
GENOPTIM	18,19,20,21,22
GENBEST	23,24,25
CREATEPACK	23
FIXPACK	24
COPY	utility
WRITEINFO1	utility
WRITEINFO2	utility
CONVRT	utility
FINDTIMES	utility
MAIN	utility

fig. 15: Program structure (continued)



a) column/row representation

CBACK
CFWD
FRSTPTR
BBACK
BFWD
PUP
PDWN

b) basic cell

fig. 16: Example of DS1 - column/row data structure

HEADBLK: In the first BLOCKK of a CT, HEADBLK records the actual size of the CT i.e. 2, 4, 8, 16, etc.

HEADCODE: HEADCODE is the internal coding for these sizes e.g. 2:1 -> 1, 4:1 -> 2, etc.

TOTSET: TOTSET gives the set of inputs represented by the CT.

COLID: An integer giving the column identification of the column in which this BLOCKK appears.

BLKARR: An array of dimension two giving the two inputs represented by this BLOCKK.

LEVID: The odd RP of the two RPs covered by this BLOCKK.

DONE: Auxiliary variable.

Note that HEADBLK and HEADCODE are zero and TOTSET is null for tree nodes which are not heads of CTs (see fig. 16 for description of heads).

The DS2 (common candidate data structure) is a doubly linked list of records. COMBF and COMBB form the links. The remainder of record COMBLK is as follows:

CCOMSET: Set of inputs which form the original

CC.

CCOLMNSET: Set of column numbers over which the original CC applies

CCOLARR: The array representation of CCOLMNSET.

SIZE1,SIZE2: Size of the CC for priority purposes and number of columns over which the CC applies.

COMB1,COMB2: Auxiliary variables.

The distinction between original CC size and CC size for priority purposes is necessary. If the original CC covering the original number of columns will not pass the commonality tests then the appropriate variable SIZE1 or SIZE2 must be reduced and the priority position changed; CCOMSET and CCOLMNSET remain unchanged. Subsets of CCOMSET which apply over subsets of CCOLMNSET are also CCs. Hence, combinations of inputs from CCOMSET of size SIZE1 and combinations of columns from CCOLMNSET of size SIZE2 may be tested for commonality.

Parts of the DS3 data structure (tree data structure) are shown in figs. 17 and 18. PATHSTA, BESTSTA, LISTPTR1, LISTPTR2, and LISTPTR3 will be described later after DS4 is introduced. The remainder of DS3 follows:

TREEID: Integer which identifies which MOMUX

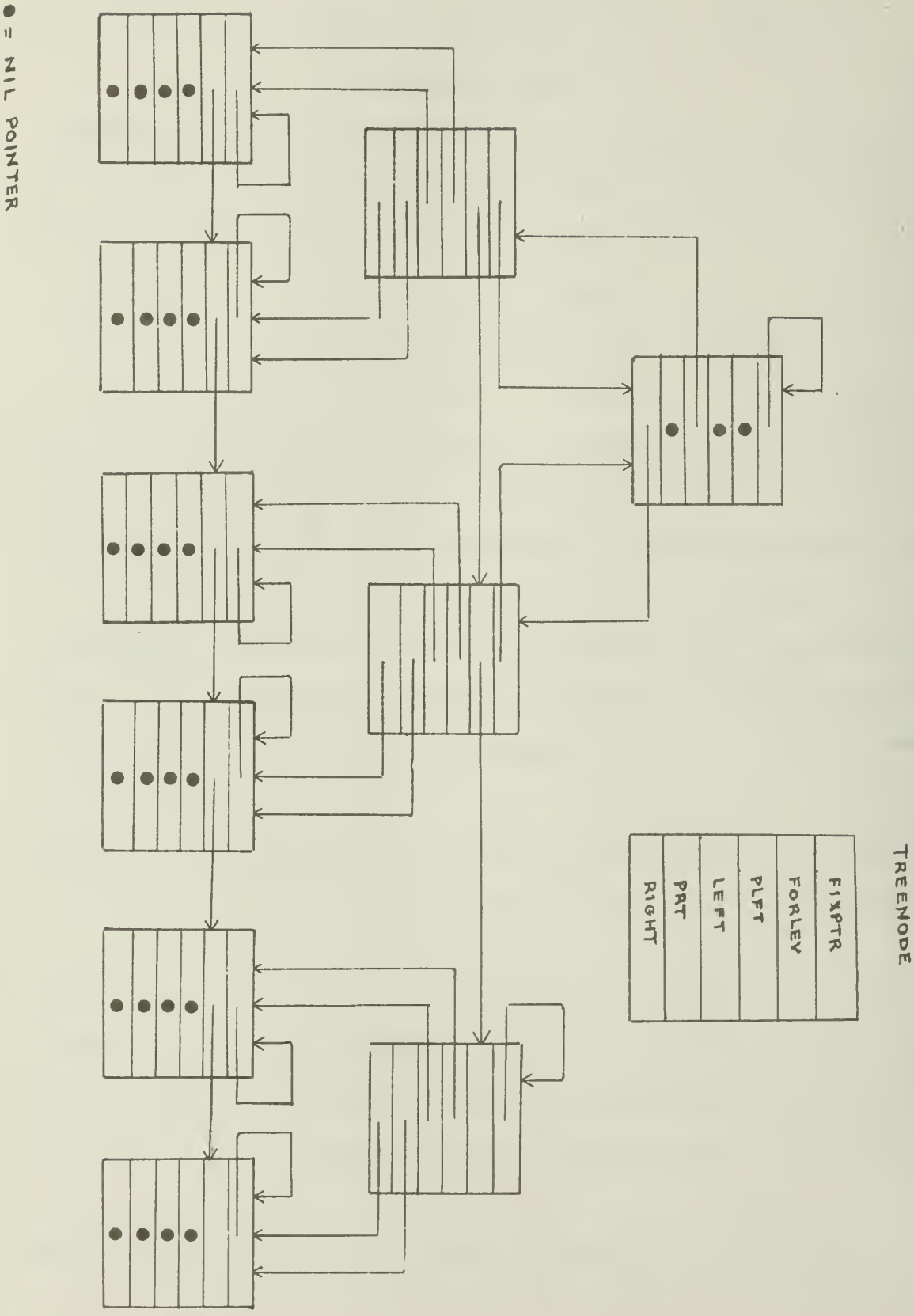


fig. 17: Example of DS3 - tree data structure

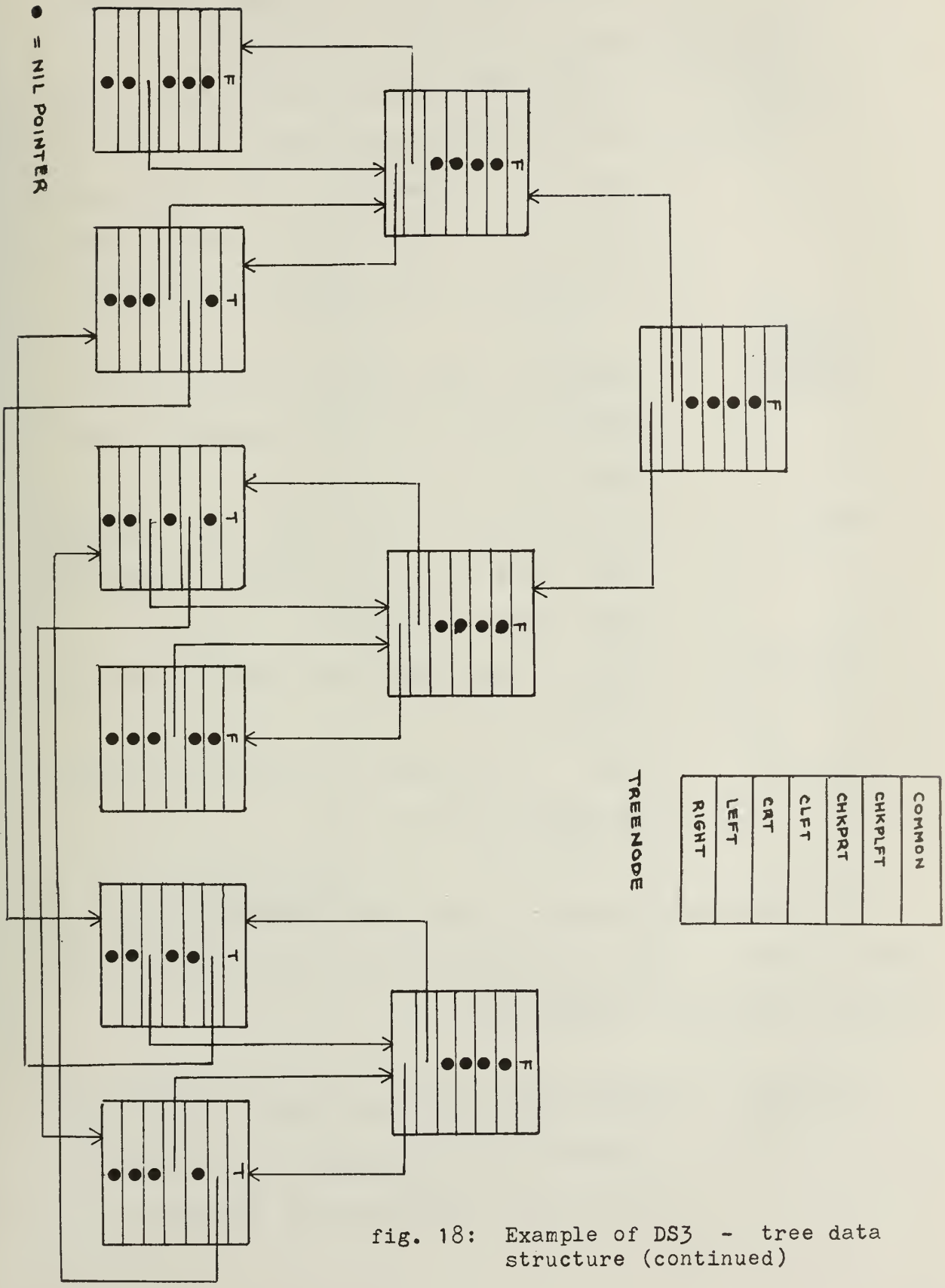


fig. 18: Example of DS3 - tree data structure (continued)

output that this tree node is associated with.

INDEX1: Integer which identifies this node. Its value can be determined by doing a postfix traversal of the tree and numbering the nodes consecutively starting at 1 for the root of the tree.

RTNUM, CENTNUM, LFTNUM: Integers referring to indices in the tree. They aid in searching for subtrees.

LEV: An integer giving the tree level for the node. See labelling in figs. 17 and 18 for examples of levels.

SZECODE: Same as HEADCODE of DS1 for the head of a CT.

BASELEV: This is the lowest tree level occupied by member nodes of a CT

The DS4A data structure (path data structure A) is shown in fig. 19. In addition to the fields in the PATHSTAT record are the following:

HEADPATH: This points to the node in DS3 which represents the head node of the first CC in the path represented by PATHSTAT.

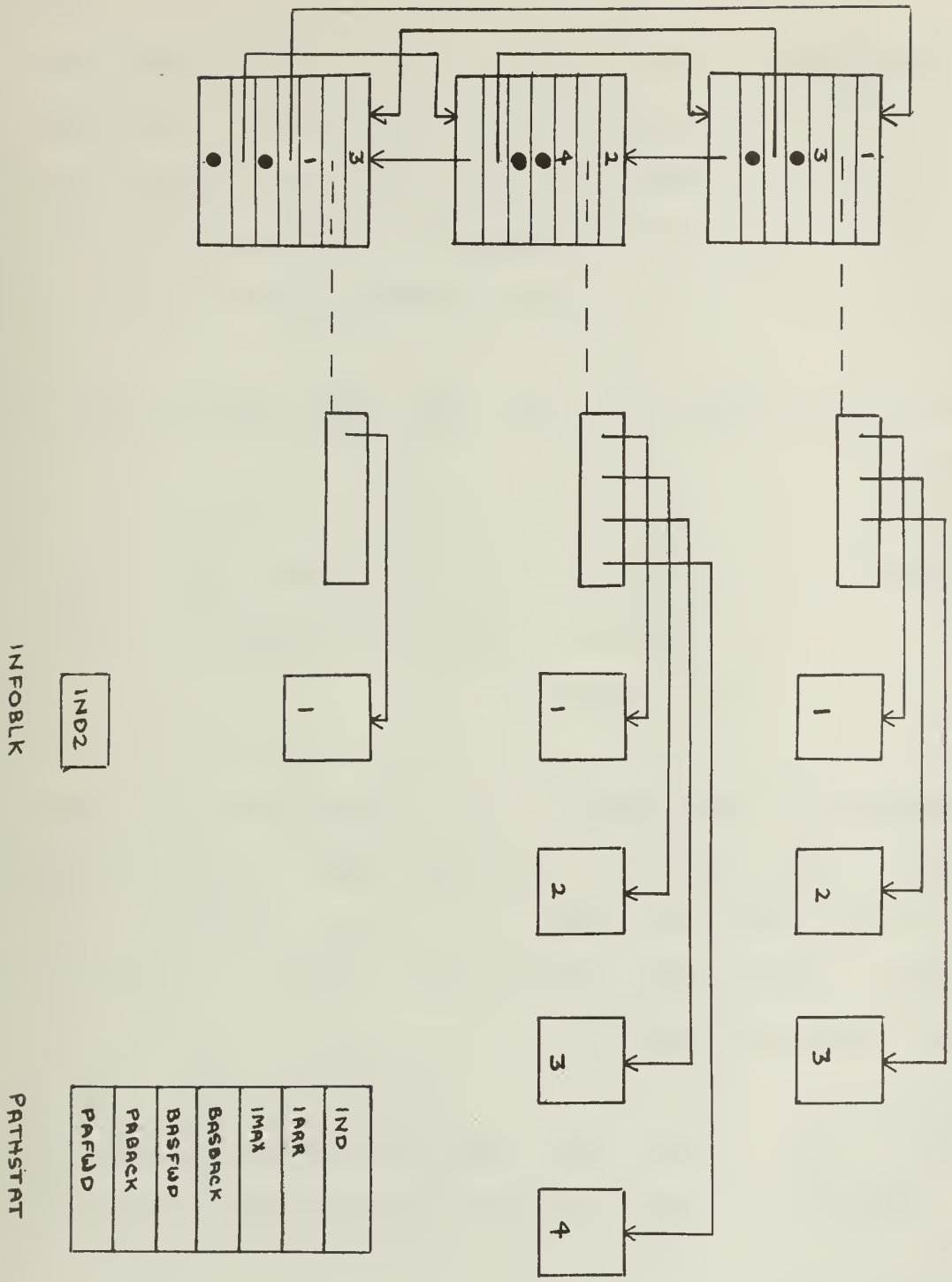


fig 19: Example of DS4A - path data structure A

PCOST,PPOW: Mnemonics for the cost of a path and power of a path respectively. PCOST and PPOW are the totals of the costs and powers of individual CTs on a path. See 1.4 for a description of paths.

The second record making up the DS4A data structures is INFOBLK.

IDTYP,INDX: These are the CT number and indexed component number associated with one CT on the path.

The DS4B and DS4C shown in fig. 20 data structures are similar to DS4A with no PCOST and PPOW. (These are not needed since DS4B and DS4C are used primarily for information storage after OP has been executed). HDPATH has the same meaning as HEADPATH.

Fig. 21 and 22 are more appropriately discussed in 2.10. Returning to those fields of DS3 not described above we have:

PATHSTA: Points to the PATHSTAT record for the path which this CT is on

OPTIMSTA: Points to the OPTIMNODE record for this

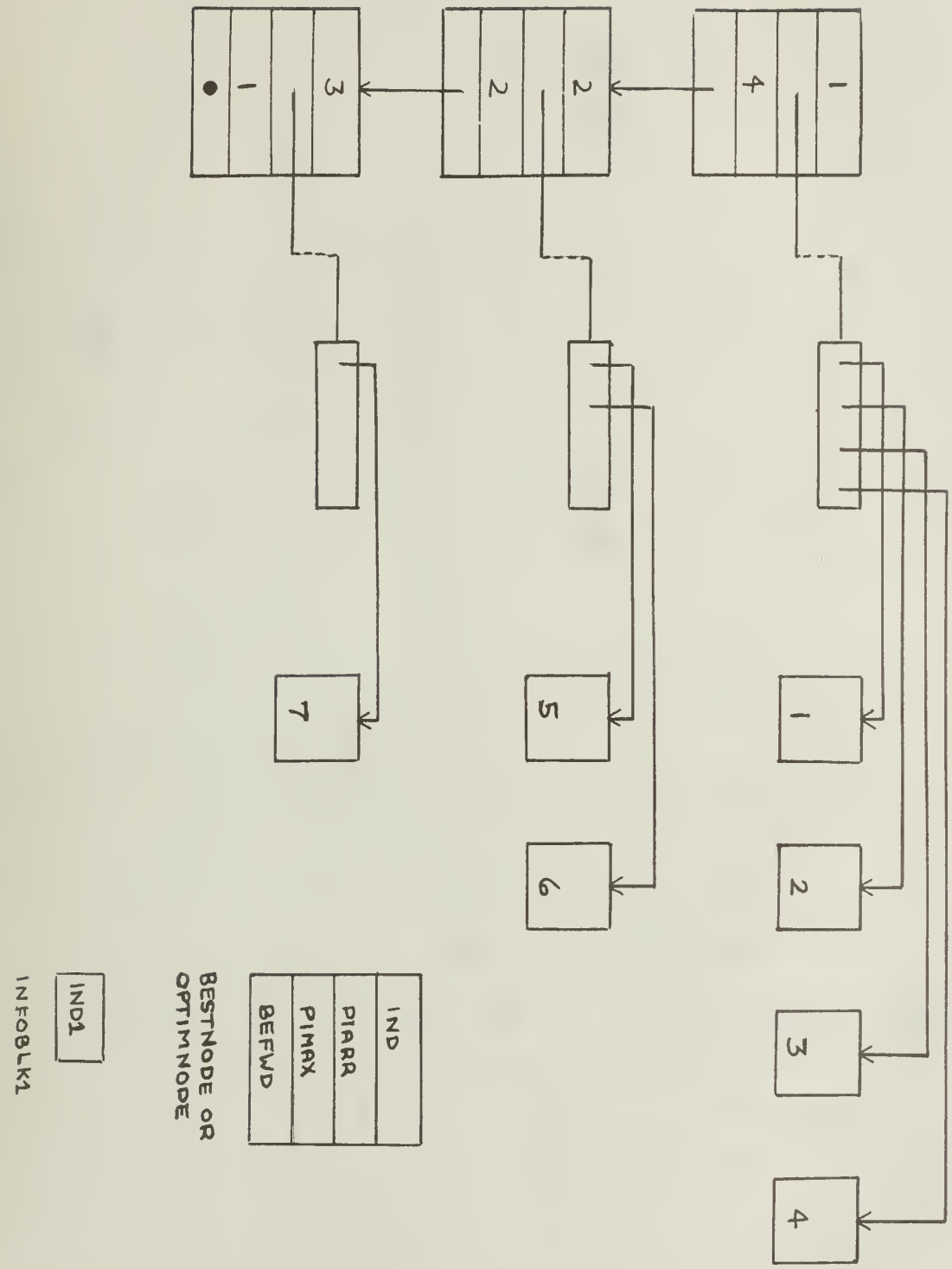
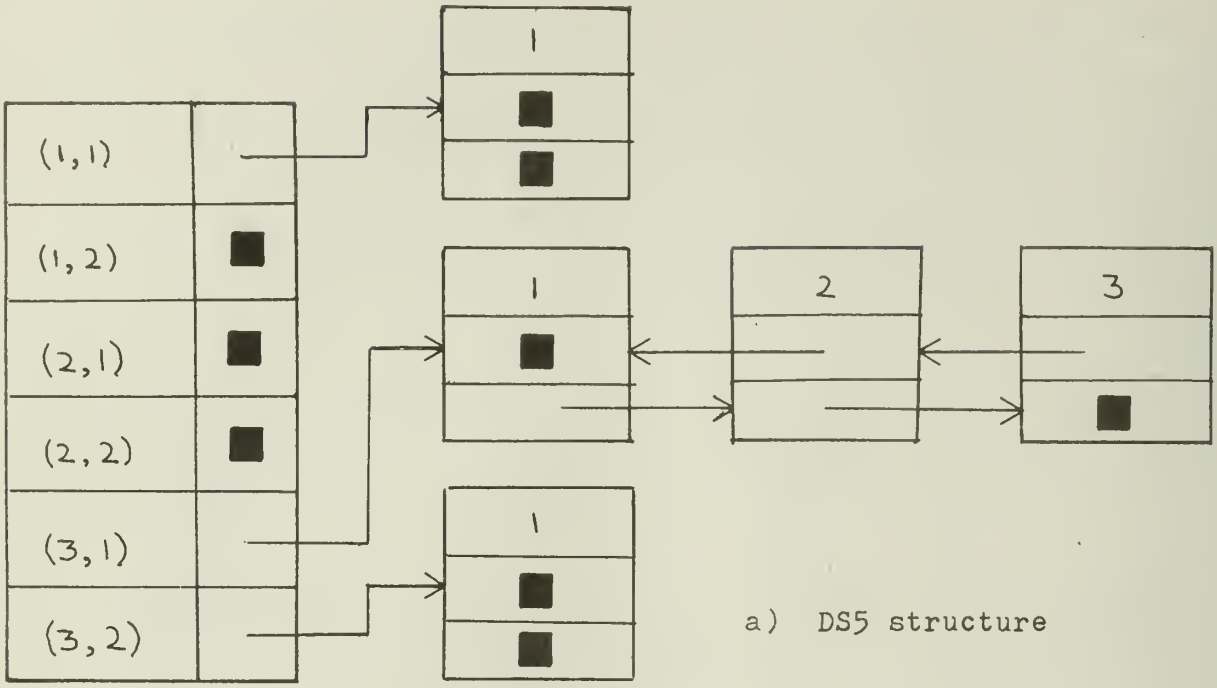
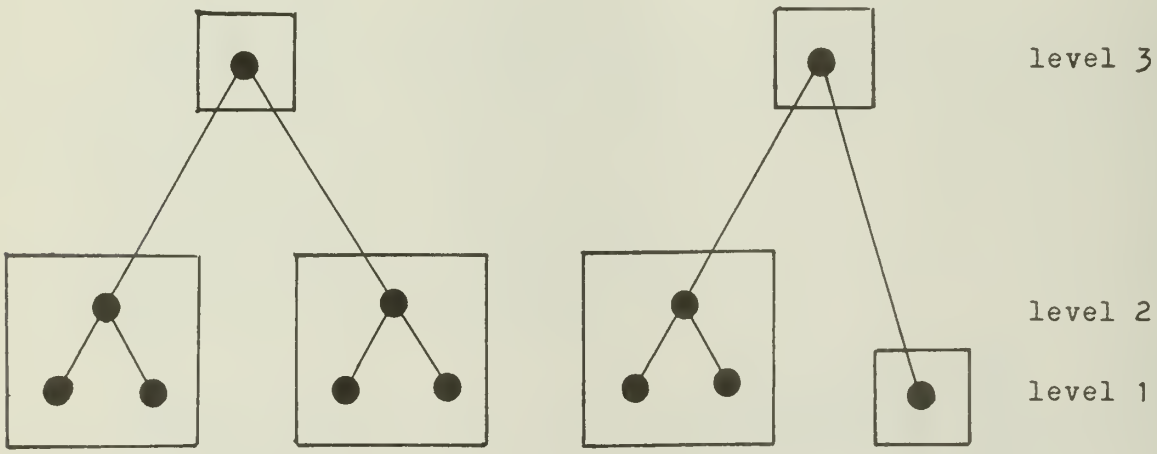


fig. 20: Example of DS4B and DS4C - path data structures B and C



a) DS5 structure



b) corresponding DS3

INDXX
PACKB
PACKF

c) PACKNODE record

fig. 21: Example of DS5 - package data structure

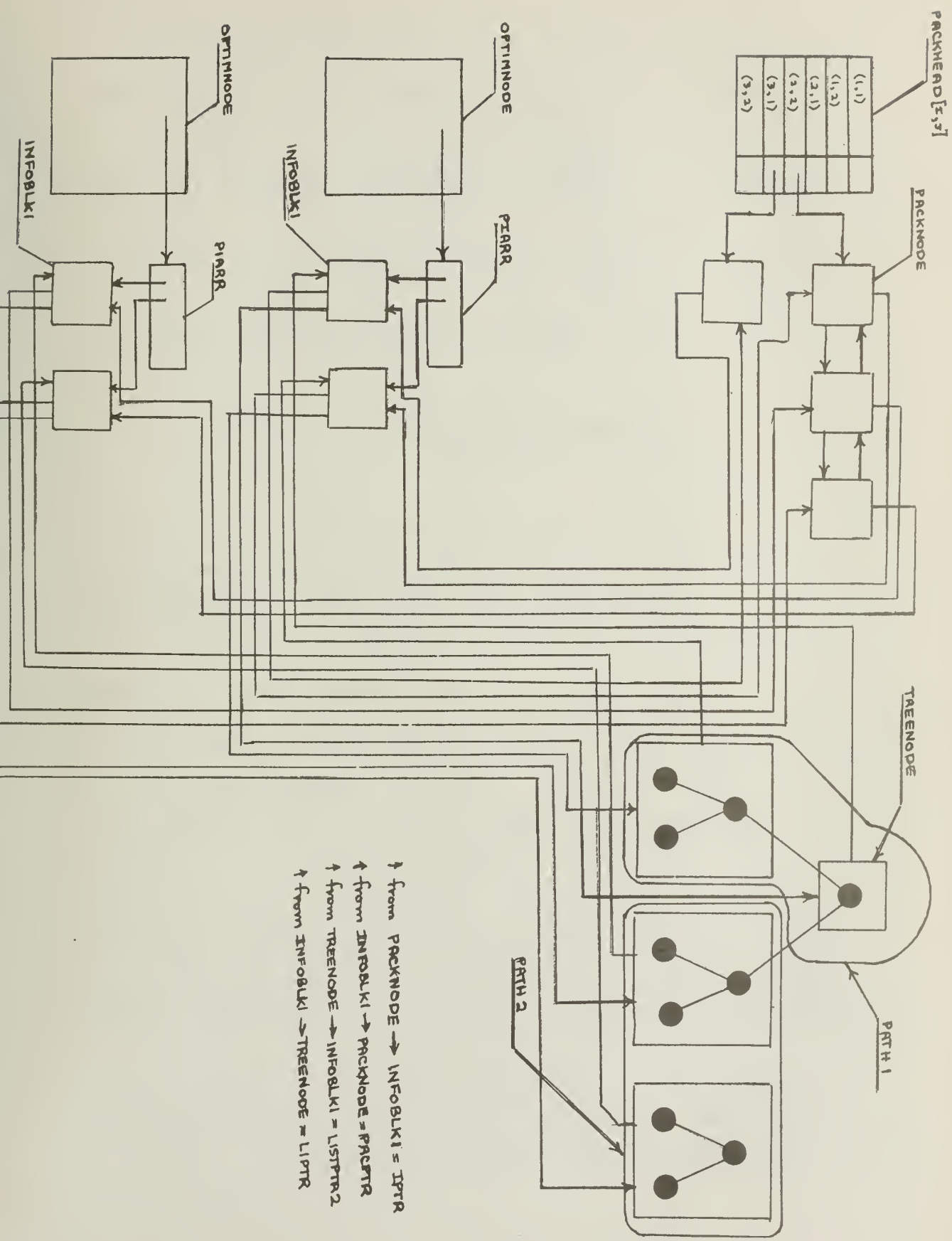


fig. 22: Example of relationship among DS3, DS4B, and DS5

path.

BESTSTA: Points to the BESTNODE record for this path.

LISTPTR1: Points to the INFOBLK record for this CT.

LISTPTR2: Points to the INFOBLK1 record in DS4B.

LISTPTR3: Points to the INFOBLK1 record in DS4C.

2.3 Generation of common candidates

Procedure GENCOL generates the DS1 representation of data read in by READINFO. Procedure GENSET then generates all CCs and calls procedure CREATECOM to construct DS2. GENSET intersects sets of inputs corresponding to columns (COLSETs) in DS1 by first intersecting n columns (n is the number of MOMUX outputs), then all combinations of n - 1 columns, etc., until all combinations of two columns are intersected. Procedure GETCOMB performs the task of generating all combinations. Intersection sets must have cardinality greater than two to qualify as a valid CC. Combinations are formed by varying higher numbered columns first.

2.4 Commonality checking

The work of commonality checking is done within procedure INTERSECT. Two interpretations of size are introduced. Before any commonality checking takes place a CC has an input set size (ISS) and column covering size (CCS) which were determined by GENSET. During the process of checking, it may be found that a VCC cannot be constructed for the ISS and CSS. First the CCS is reduced. Note that although the ISS or CCS may be reduced the sets from which the choices are made (CMS, commons set and CLS, column set) remain unchanged. If no further reductions in CCS are possible, the ISS is reduced until the ISS reaches 2.

If a VCC is not found as a result of tests 1 to 5 for a CC, variable FNDONE is false. If the combinations of columns of size CCS have not been exhausted, a new combination is determined by procedure GETCOMB. If exhausted, SZECNST (size constraint) is set true if ISS = 1 (CT is 2:1) and CSS = 2. If SZECNST is true the CC entry is deleted using procedure DELCOM. DELCOM is called to remove the old record for this CC, since the priority and hence ordering of the CC in DS2 must be changed. Then CREATECOM is called to place the CC back in DS2 at the appropriate priority level.

If FNDONE is true, then the CC entry is examined to see if a smaller CC covering the same number of columns or the same size CC covering fewer columns or both can be found. The old CC is deleted by DELCOM and possible new CCs are added depending on the results of tests on the old CC entry.

The commonality tests are performed by procedures FINDSETS, CHKLVLS, CHKPLC, and OKCHK. Prior to checking, procedure GENLVLS sets up the testing environment. Two important variables are initialized. Each has a particular value corresponding to a combination of each possible CT, each column of DS1, and each possible position within that column which can be occupied by a CT. These positions are valid column positions (VCPs) for a certain CT. Variable LEVSETS adds up the number of columns at each RP in DS1 for all the RPs in a VCP and forms an array of these ordered by size of sum. Variable LEVVSETS is the set of RPs which correspond to a VCP (see fig. 23).

Procedure INIT forms an array of sets called SELSET. The first index of the array is the column involved and the second index corresponds to RP2 as shown in fig. 25. Each set in SELSET corresponds to the inputs in the CC which are also contained within a DS1 record at the RP2 indicated by the second array index. Thus each SELSET set has at most two elements.

****

sum of columns	column number 1	2	3
3	A	U	M
3	B	V	N
3	C	W	O
3	D	X	P
3	E	Y	Q
2	F	Z	
2	G	L	
1	H		
1	S		
1	T		

```
LEVSETS[1,1,1] = 0020000000
LEVSETS[1,1,2] = 0020000000
LEVSETS[1,1,3] = 0110000000
LEVSETS[1,1,4] = 1100000000
LEVSETS[1,1,5] = 2000000000
    other columns and CTs are similar
LEVVSETS[1,2,1] = {1,2,3,4}
LEVVSETS[1,2,2] = {5,6,7,8}
    other columns and CTs are similar
```

i -> column number
j -> CT number e.g. 1 -> 2:1, 2 -> 4:1, etc.
k -> integer giving position in column

Letters have been used in place of numbers for entries in DS1 to give greater clarity.

fig. 23: Demonstrating LEVSETS[i,j,k] and LEVVSETS[i,j,k]

****

The RRPM (see 1.3) for the inputs in each SELSET entry is determined in procedure FINDSETS. The RRPM of each SELSET is set variable RANGE. The values of RANGE for each column are intersected to determine the overall RPPM, represented by variable CHCSET. SELSETS having the same RRPM are assembled together in variable CHCARR.

Given a CC with a certain ISS, procedure CHKLVLS forms combinations of inputs (CHKSET) from the CMS of size ISS and for each such combination checks combinations of VCPs of CTs of size ISS taken from a combination of columns of size CSS formed in INTERSECT. Variable FNDONE in CHKLVLS is true if a particular combination of VCPs passes test 1 using the results of procedure FINDSETS. Test 1 is as follows:

> For each column in the combination of columns from INTERSECT and for a particular VCP under test within that column, variable TSTSET is set equal to LEVVSETS with appropriate array indices for this VCP. Then the intersection of CHKSET and CHCARR for each grouping of SELSETS is set equal to CHKK. Immediately generate a negative result is the intersection yields an odd number of set elements or a number of elements less than two. Assuming the above tests were positive, test to see if the cardinality of the intersection of TSTSET and CHCSET with appropriate array indices is equal to

the cardinality of CHKK. Since CHKSET and CHCARR are sets of inputs and TSTSET and CHCSET are sets of RPs, test 1 basically tests for the existence of a sequence of inputs and a corresponding sequence of RPs which are both consistent the the results of procedure FINDSETS. (See fig. 5 for a demonstration of test 1).

If test 1 is passed for a combination of VCPs covering columns specified by INTERSECT, then procedure OKCHK, where tests 2 and 3 are performed, is entered (see fig. 6). Two critical variables in OKCHK are TESTARR which adds up the numbers of columns at the RP corresponding to each input in CHKSET and puts the results in the same format as LEVSETS, and LVLCHK which forms sets of inputs, each set having the same sum of columns at the RP corresponding to that input. Test 2 is positive if TESTARR equals LEVSETs for each VCP in the combination (see fig. 24). If test 2 is passed, then two cases for test 3 are defined as follows: 1. for all pairs of columns specified by INTERSECT, if the VCP for both columns is the same then the inputs at each RP in these columns must match, or 2. if the VCPs don't match, then by looking at LVLCHK for each column in the pair, we may determine if the same ordering of the inputs in each VCP will produce an ordering of sums of columns at each RP which would be consistent with LEVSETs for each VCP. The details are involved, but an inspection of the code in this portion

****

```
                    RP

                    -------------
                    1           x
                    2           x
                    3     8     x
                    4     11    x
                    -------------
                    5     1     x
                    6     2     x
                    7     5     x
                    8     6     x
                    -------------
                    9     7
                    10    9
                    11
                    11
                    -------------
```

The above gives a negative test 2 result.

```
                    RP

                    7
                    8
                    -------------
                    9     8     x
                    10    11    x
                    11    1     1
                    12    2     x
                    -------------
                    13    5     x
                    14    6     x
                    15    7     x
                    16    9     x
                    -------------
                    17
                    18
```

The above gives a positive test 2 result.

fig. 24: Demonstration of test 2

****

of OKCHK should make the process clear.

Assuming test 3 is passed, procedure CHKPLC will perform tests 4 and 5 (see figs. 7 and 8 for examples). Now a new data structure having CREFF records is described. There is a CREFF record for every RP2. Field OVERALL is a set which includes RPs covered and columns covered by all inputs contained in any VCC which has any input at the RP that indexes the array of CREFFs. Field INDIV is an array of COMREFREC records, one for every column of DS1. Each COMREFREC contains field CMMNS which records the set of RPs covered by the inputs contained in the VCC at the column number which indexes INDIV. FREESET is the set of inputs making up the VCC at the column number which indexes INDIV. (See fig. 25 for an example).

Test 4 is performed for each column specified by INTERSECT:

> An RP2 is determined for the CC under test and the OVERALL field at the CREFF indexed by this RP2 is checked. If OVERALL is empty then testing for this column is completed. Otherwise, each non-empty INDIV record is checked. The inputs in the column which indexes INDIV and which correspond (have the same RP) to the inputs contained in the CC are determined. The

RP2	RP	column number			
		1	2	3	4
1	1	17	9	x	x
	2	18	8	x	x
2	3	x	10	x	x
	4	x	15	x	x
3	5	x	x	x	x
	6	x	x	x	x
4	7	x	x	x	x
	8	x	x	x	x
5	9	x	x	x	x
	10	8	x	x	x
6	11	10	x	17	17
	12	15	x	18	18
7	13	x	x	9	
	14	x	x	8	
8	15	x	x	10	
	16	x	x	15	

```
COMREF[6].OVERALL = {9,10,11,12} + {13,14,15,16}
     + {1,2,3,4} + {1,2} + {11,12} + {11,12}
                 = {1,2,3,4,9,10,11,12,13,14,15,16}
COMREF[6].INDIV[1].CMMNS = {9,10,11,12} + {1,2,3,4}
     + {13,14,15,16}
COMREF[6].INDIV[1].PLCMNT = {1,2,3}
COMREF[6].INDIV[1].FREESET = {9,8,10,15}


COMREF[3].INDIV[1].CMMNS = {1,2} + {11,12} + {11,12}
COMREF[3].INDIV[1].PLCMNT = {1,3,4}
COMREF[3].INDIV[1].FREESET = {17,18}


x = unspecified input
+ = set union
{x,x,x,x} = set
```

fig. 25: Examples of entries in the CREFF and COMREFREC records

inputs so determined are actually part of a VCC previously tested. To see if the inputs of the previously tested VCC may be regrouped or reordered or both to meet the requirements of the CC under test, procedure OKCHK is called with parameters which reflect the new grouping and ordering.

Assuming test 4 has been passed, the second half of procedure CHKPLC performs test 5. For every pair of columns, say I and J, specified by INTERSECT, the OVERALL sets are intersected. If the intersection is non-empty, there are at least two columns, say P and Q, which have inputs which are contained in VCCs, say in columns M and N, which are at the same RPs as the CC in columns I and J. Thus for the ordering of the CC to be consistent in columns I and J, consistency between P and Q must be checked. Then the RPs of each input of the CC in columns I and J is determined. The inputs, say X and Y, in columns M and N corresponding to these RPs are then determined. Then the RPs corresponding to X and Y in columns P and Q are determined. Test 5 is passed if either of two conditions is satisfied, either the RPs of the Xs and Ys generated by all inputs in the CC in columns I and J are the same, or the sets of Xs and Ys are disjoint. Again, the above two conditions must be satisfied by every pair of columns where the OVERALL set is non-empty.

If test 5 has been passed, procedure PLC is called to adjust the CREFF and COMREFRECs to show the new VCC. Then procedure MOVEM is called to adjust DS1 to show the new VCC.

2.5 Tree generation

Procedure MAKETREE generates the DS3 data structure in the form of binary trees of 2:1 CTs. DS3 is generated as if each tree in a MOMUX were an independent entity. Commonality and the superposition of CTs is considered in 1.5. Trees are built by linking up subtrees having numbers of leaves that are powers of two. The variable EXSINP represents the number of inputs in a column of DS1 not yet built into trees. The largest power of two which is less than or equal to EXSINP is the number of leaves that the next tree to be generated will have. Procedure LOGOF finds this largest power of two. Procedure PERFTREE generates subtrees recursively. Procedure NEWNODE is called by PERFTREE when new DS3 node records must be formed. When no new power of two trees can be formed, the ones already formed are linked together as shown in fig. 26. The root of the tree is passed as a parameter back to the MAIN procedure where it becomes a variable called TREEHEAD indexed by the column number corresponding to the tree. The node numbering of every 2:1 CT of a MOMUX is unique. Thus

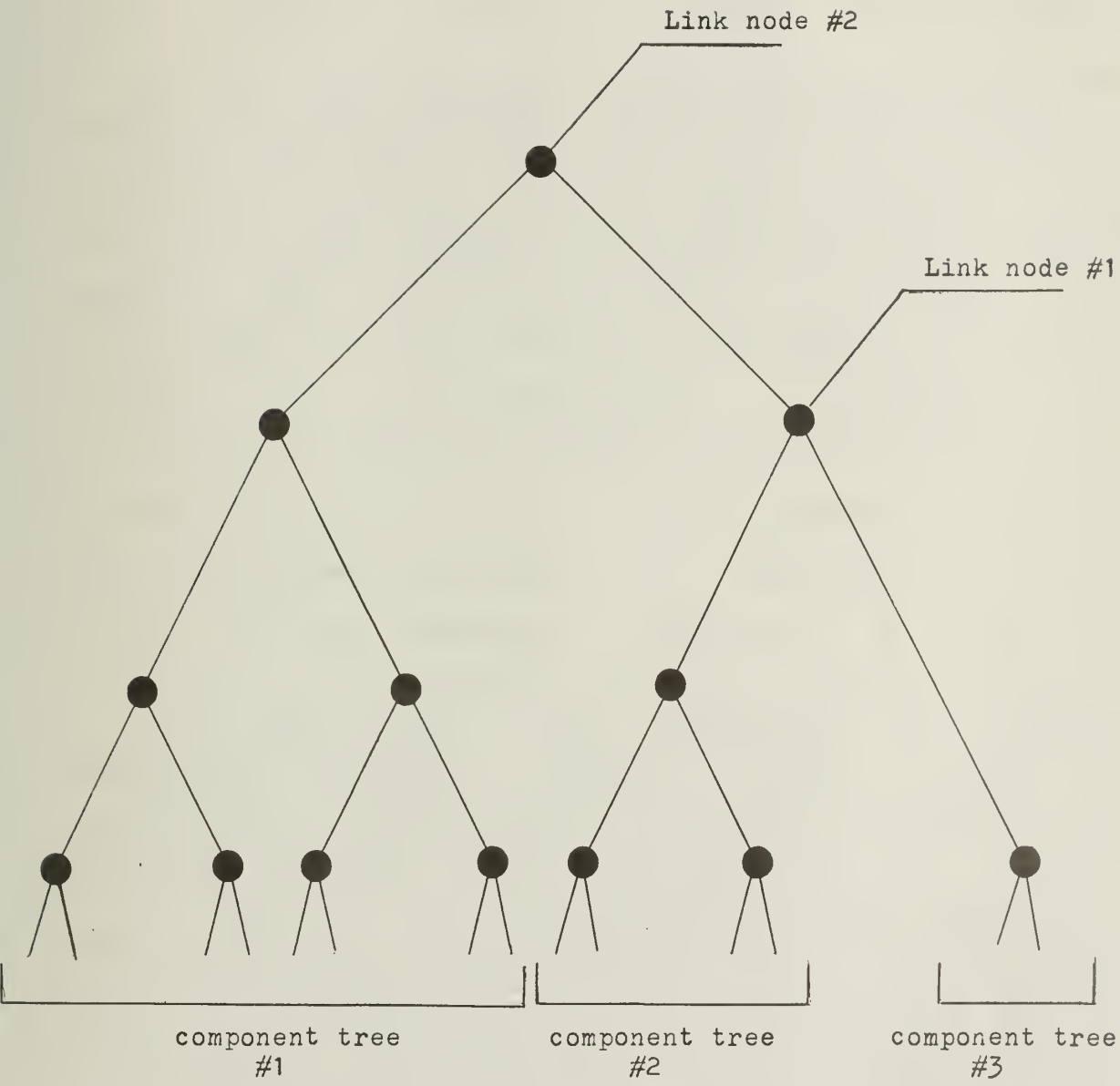


fig. 26: Generation of DS3 data structure

at each invocation of MAKETREE, the updated node count is passed back to MAIN.

2.6 Component type superposition

Procedure PACKS finishes the translation from the DS1 data structure to the DS3 data structure and adds to this some new information, namely the CTs that may be necessary to join together CTs representing blocks of inputs. PACKS is called once for each tree in a MOMUX, the tree number is given by parameter I. Since there is a one to one correspondence at this point between the leaves of the trees in DS1 and the columns of DS1, the positions of the roots of subtrees which represent CTs corresponding to groups of inputs may be determined. SUPERPACK performs this determination returning the root (POINT5) and a pointer (POINT4) to a node having a LEFT or RIGHT pointer to the root. Using this information and accounting for membership of the CT in a VCC, pointers PLFT, PRT, CHKPLFT, CHKPRT, CLFT, and CRT are initialized (see figs. 17 and 18). Procedure MARK is then called to fill in fields PACKIND, FIXPTR, SZECODE, and BASELEV of each DS3 node in the CT. Variable PKINDX is incremented whenever a CT is placed in DS3; each CT in DS3 is thus assigned a unique number (PACKIND). When a VCC is detected, the above calls to

SUPERPACK and MARK are performed for each tree where the VCC appears.

The joining CTs shown in fig. 27 are determined by using procedures FINDELE and FINDMAX. Procedure FINDELE finds the first node in a tree which has not already been assigned to a CT by doing a postfix traversal and returns a pointer to it (POINT4). Then by trying successively larger CTs, the largest possible CT is found by procedure FINDMAX. MARK is then called to fill in the rest of the DS3 information. Calls to FINDELE and FINDMAX continue until all trees in a MOMUX are completely covered by CTs.

2.7 IAB determination

IABs (see fig. 9) are determined in procedure RELATIONAL. Variable INTERACT, formed in PACKS and referenced by column number, gives the set of columns covered by VCCs having a CT in the referencing column. Thus IABs can be formed by taking the largest disjoint sets among those in INTERACT. Procedure INCL performs the test for disjoint sets. By interatively taking the largest set among those in INTERACT which is disjoint from any set already selected to be an IAB, new IABs are formed. IABs are represented by a two dimentional array, INTERARR. The first

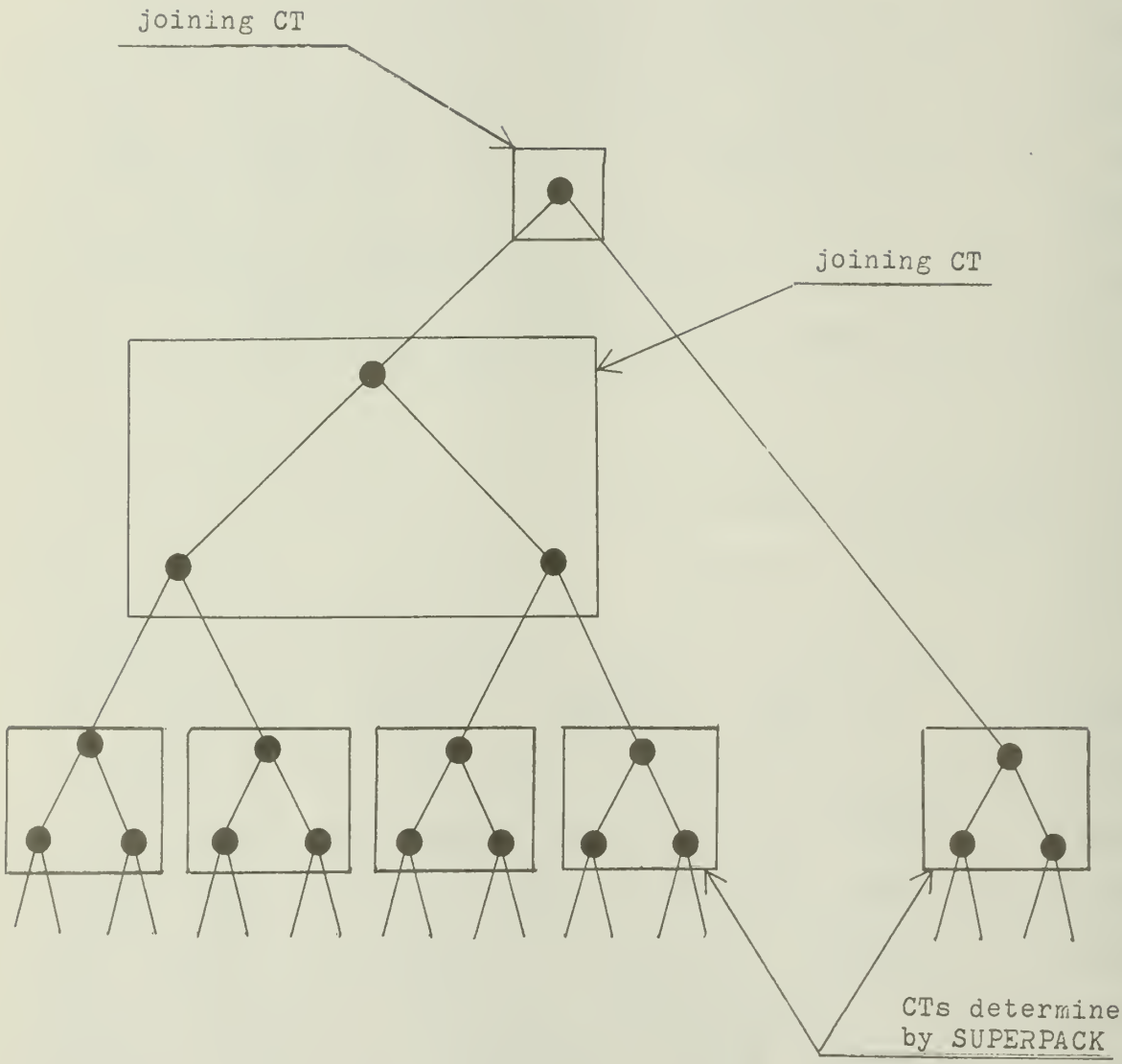


fig. 27: Example of CT superposition showing joining CTs

index gives the IAB number, the second index gives successive column numbers contained in the IAB.

2.8 Path generation

Procedure CPATHNONCOM generates all paths not containing a CT which is a member of a VCC. The three parameters of CPATHNONCOM are POINT, which gives the head node in DS3 representing the CT about to be entered into a path; NEWPATH, a switch variable indicating whether the CT pointed to by POINT is to begin a new path or not; and I, a path index variable. If a new path is indicated, then procedure CNEWPATH creates a new PATHSTAT, OPTIMNODE, and BESTNODE record and initializes most fields involving pointers. Procedure FILLINFO then fills in the remaining information in the new record. The DS3 node pointer, POINT, is then advanced to the leftmost node still within the CT. A flowchart for CPATHNONCOM is shown in figs. 28, 29 and 30. PASSPTR1 and PASSPTR2 are pointers to the right and left subtrees emanating from POINT (if these subtrees exist i.e. POINT does not point to a leaf of DS3). If a left subtree exists and the present branch is the leftmost branch and the left subtree is not a CT which is a member of a VCC then CPATHNONCOM is called with NEWPATH not set. If a left subtree exists and the present branch is not the leftmost

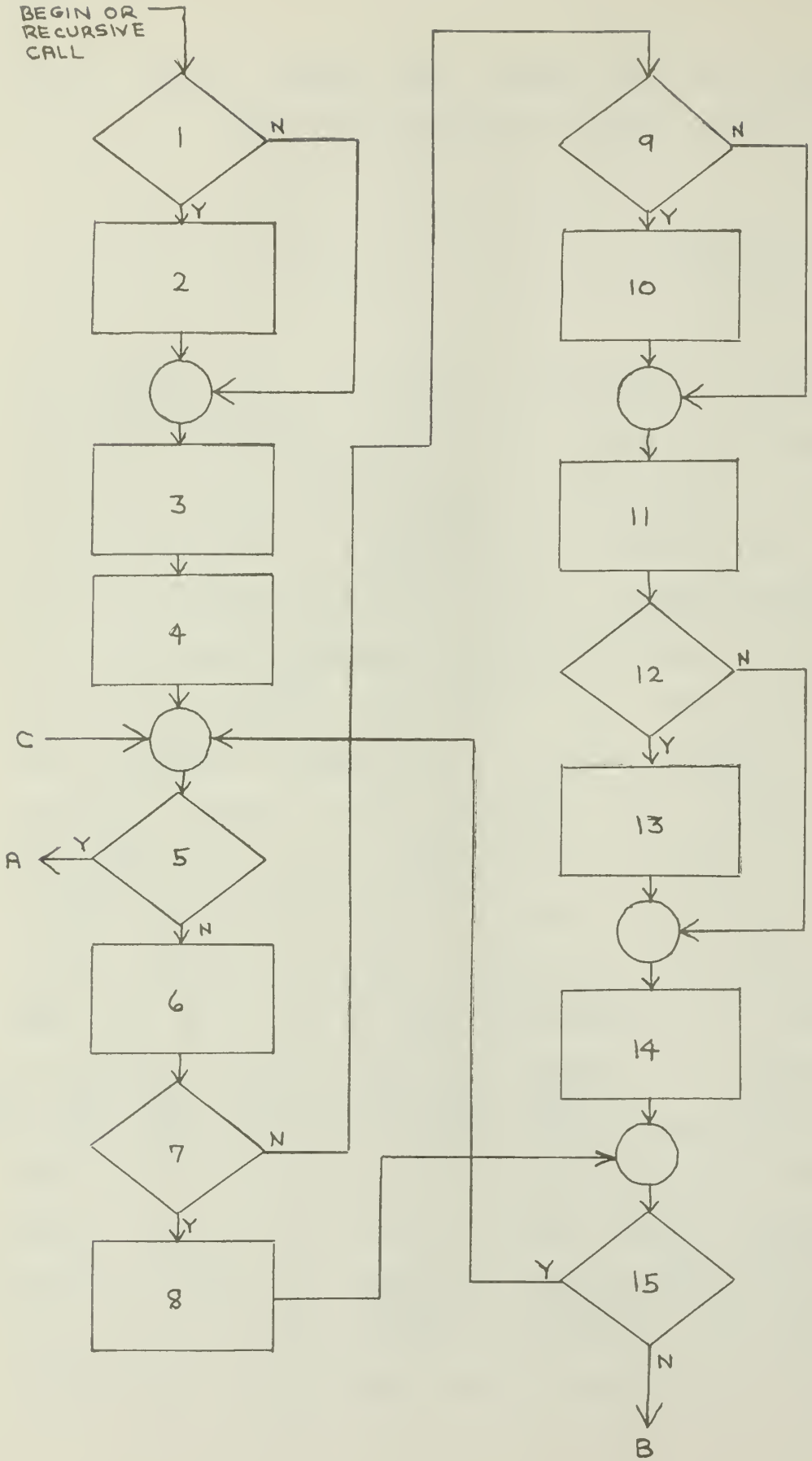


fig. 28: Flowchart for procedure CPATHNONCOM

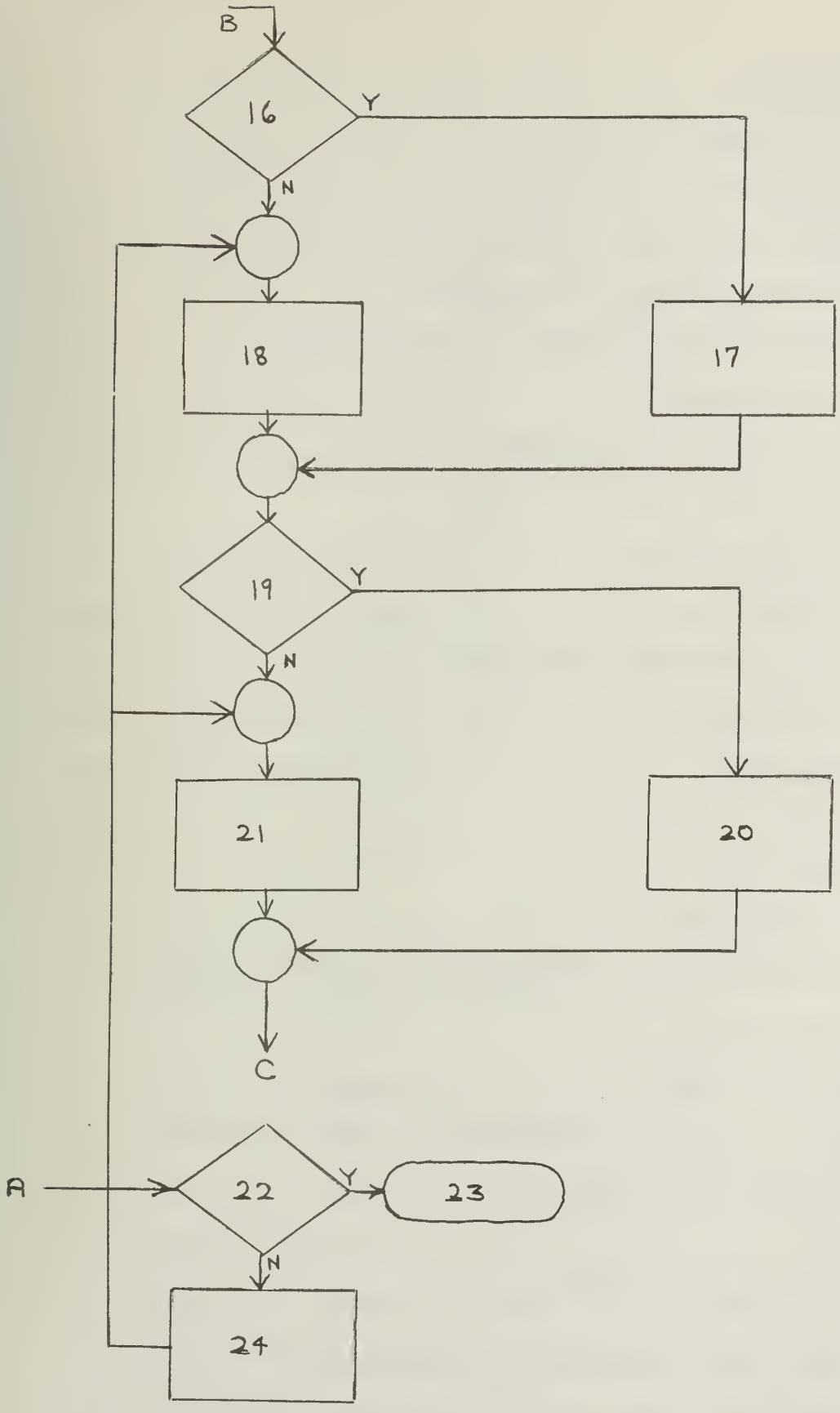


fig. 29: Flowchart for procedure CPATHNONCOM (continued)

1. new path?
2. call CPATHNONCOM
3. call FILLINFO
4. advance pointer to leftmost node of CT
5. branches from this CT exhausted?
6. update pointer to point to a new branch
7. at a leaf of DS3?
8. set switches LCHK and RCHK
9. is CT to left part of a VCC?
10. set switch LLCHK
11. set PASSPTR1
12. is CT to right part of a VCC?
13. set switch RLCHK
14. set PASSPTR2
15. is LCHK or RCHK set?
16. is LLCHK set?
17. call HANDLECOM
18. recursive call to CPATHNONCOM - POINT = PASSPTR1
19. is RLCHK set?
20. call HANDLECOM
21. recursive call to CPATHNONCOM - POINT = PASSPTR2
22. recursive calls exhausted
23. end
24. return to call point

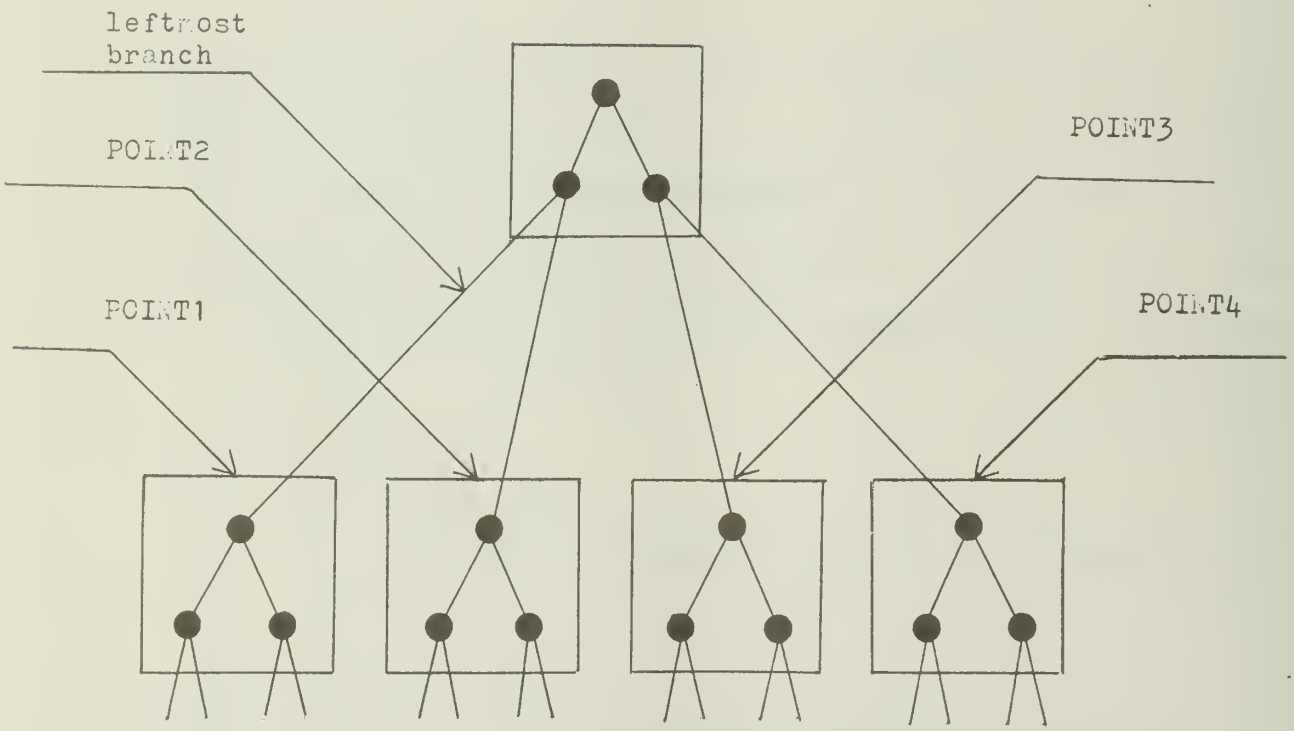
fig. 30: CPATHNONCOM flowchart key

branch and no common is involved, then CPATHNONCOM is called with NEWPATH set. If a left subtree exists and a common is involved, procedure HANDLECOM is called. The same sequence of choices exist for a subtree to the right, except that it is definitely not a leftmost branch and thus NEWPATH is always set. After returns from HANDLECOM or recursive calls to CPATHNONCOM, variable POINT is updated to point to the right of the present one (see fig. 31).

Procedure HANDLECOM checks to see if all CTs abutting any CTs which are part of a VCC have been assigned to paths. If they have not been assigned, control is passed back to the calling point of HANDLECOM; if they have been assigned, then calls to NEWPATH and FILLINFO are made to create a new path.

2.9 Optimization without package constraints

Variable PATHPRES of procedure OPTIM, a pointer to the path for which a combination of CTs meeting delay criteria is desired, is initialized to be the last path of the IAB considered previously to the present one. Three possible outcomes from procedure OPTIM are possible: 1. switch FINIGOOD is set meaning that a combination of CTs covering the entire IAB and meeting delay criteria has been found; in



Variable POINT takes on values POINT1, POINT2, POINT3 and POINT4

fig. 31: Updating variable POINT

this case PATHPRES will point to the first path in the next IAB, or 2. switch FINIBAD is set which means that no possible combination of CTs meeting delay criteria can be found and PATHPRES points to the last path of the IAB previously considered, or 3. the CPU time allocated for finding a pseudo-optimum solution for this IAB has been exceeded (see 3.5) and FINIBAD is set. A flowchart for OPTIM is shown in figs. 32, 33 and 34.

In procedure MOVPTR, an SP is indicated by pointers BASFWD or BASBACK not nil. Procedure INCIND updates the INDEX field of the INFOBLK records which make up a path. In procedure CHGTYPE variable TOPDLY is used to track the current total delay from the root of the tree (delay at the root is arbitrarily set to be the delay specified for the individual trees of a MOMUX). In the case of paths terminating at leaves of the DS3 data structure, TOPDLY is compared with the input delay (BOTDLY). If TOPDLY is greater then or equal to BOTDLY then the selected indexed components meet the delay criteria.

When OPTIM is terminated with FINIGOOD set, the merit factor without package constraint is calculated in GENOPTIM by adding up the total cost and total power of each path making up the IAB and calling procedure MERIT with a dummy value (DUMINT) substituted for the package constraint

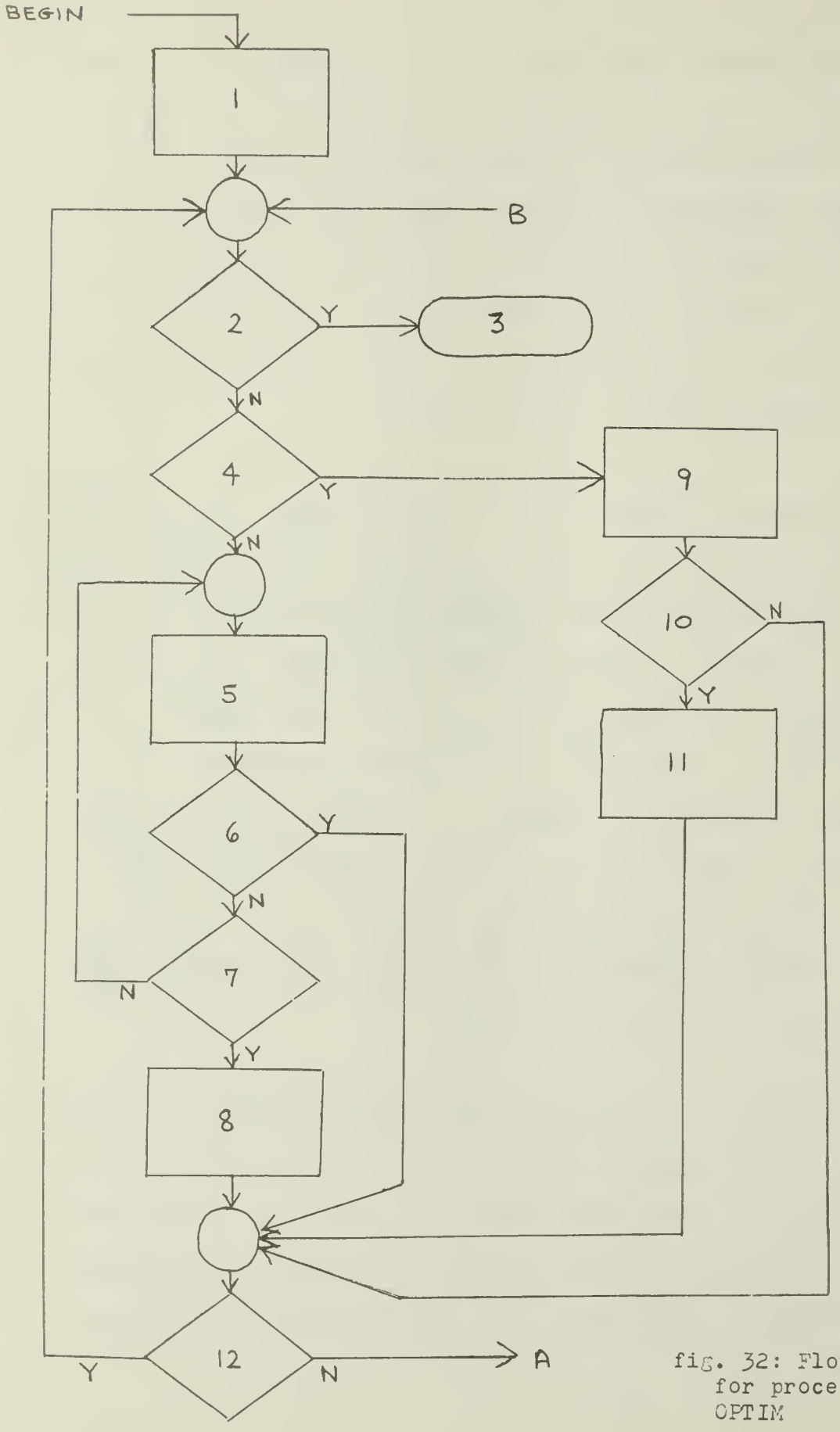


fig. 32: Flowchart for procedure OPTIM

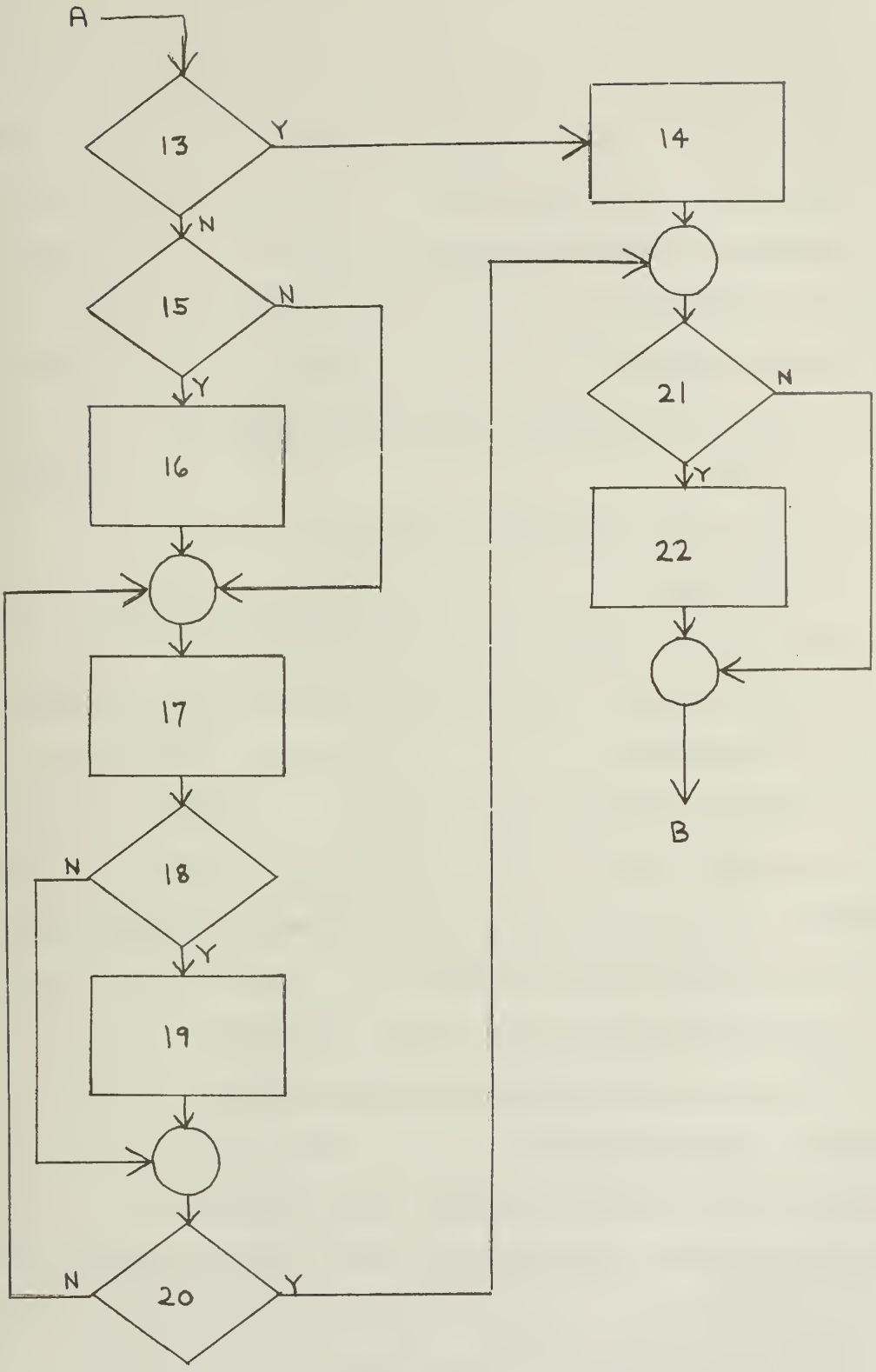


fig. 33: Flowchart for procedure OPTIM (continued)

1. initialize path pointer
2. FINIGOOD or FINIBAD set?
3. terminate
4. FOUND true?
5. move path pointer one position backward
6. at an SP?
7. at last path in previous IAB?
8. set FINIBAD
9. move path pointer one position forward
10. is path pointer pointing to first path in next IAB?
11. set FINIGOOD
12. FINIGOOD or FINIBAD set?
13. at an SP?
14. make an arbitrary selection of indexed components
15. is this CT part of a VCC?
16. consider delays of all abutting paths
17. choose another combination using INCIND.
18. meet delay criteria?
19. set switch FOUND and check merit factor
20. all possible combinations of indexed components checked?
21. exceed time allocated?
22. set switch FINIBAD

fig. 34: OPTIM flowchart key

parameter. If the new merit factor is found to be greater than the old, OPTIMMERIT referenced by the IAB number, then OPTIMMERIT is updated. At the same time the DS3B data structure is updated by changing the INDX and IDTYP fields of records INFOBLK1. Recall that the INFOBLK1 records are assembled into an array in record OPTIMNODE, one for CT in a each path. The first OPTIMNODE record is pointed to by OPTIMHEAD.

2.10 Package considerations

The array PACKHEAD is initialized to be nil in procedure GENBEST. Beginning at level FIXLEV, the next level to be fixed (see 1.5), each CT whose head node is at level FIXLEV causes a call to procedure CREATEPACK. Procedure CREATEPACK constructs the DS5 data structure (see fig. 21). The first index of array PACKHEAD gives the CT type number i.e. the index for a 2:1 is 1, 4:1 is 2, etc. The second index gives the tree level as shown in fig. 21. Procedure CREATEPACK first creates a new PACKNODE record for each occurrence of a CT in the DS3 data structure for tree levels less than or equal to FIXLEV. The placement of the record into the DS5 data structure is done using delay as a criteria. The string of PACKNODE records at a particular I, J index of PACKHEAD is traversed starting at the packnode

pointed to by PACKHEAD until a PACKNODE representing an indexed component from DS3 whose delay is greater than or equal to the delay of the indexed component about to be entered is encountered. The new PACKNODE is then inserted into the doubly linked list. Special cases such as the first PACKNODE at a particular I,J and the last PACKNODE are also dealt with in CREATEPACK.

After all the necessary calls to CREATEPACK at a particular tree level are made, FIXPACK is called. FIXPACK performs the assignment of indexed components to IC packages for each CT size of indexed components and each PACKHEAD string that is not empty. Power, cost, and packages are labelled. Included in the totals are unused indexed components within packages within packages. Note that no attempt is made to transform any data structures in an attempt to use these wasted parts of ICs.

> Assumption 8: No attempt to utilize wasted components in an IC package as a result of assignments in FIXPACK is made.

In GENBEST the power, cost, and package totals for each level in the tree are added to get an overall total. Note that the power, cost, and package totals for tree levels not considered at this invocation of GENBEST remain unchanged

from past invocations (OPTIMDATA is global). A new merit factor is calculated this time, using the package constraint by calling procedure MERIT using as the last parameter of the call the number of packages determined by FIXPACK. If the newly calculated merit factor is better than any previously calculated one, then the DS3B data structure (generated in procedure GENOPTIM) is copied to the DS4C data structure, which forms the final interface between the MDAT and the outside world via output procedures. Now that IC packages have been determined for level FIXLEV, the DS4A data structure is modified to reflect the new fixed indexed component assignments for that level. FIXLEV is decremented.

Now is the best time to comment on fig. 22 which shows the relationship among DS3, DS4B, and DS5. It may be obvious that there is a considerable amount of redundant information stored in these structures but the additional complexity may be justified by the simplified data manipulations allowed by these structures. Each structure performs an interfacing function between two other structures e.g. DS3 interfaces DS1 with DS4A, DS4B, and DS4C. DS4A interfaces DS3 with DS5. DS3 is specialized for CTs. DS4A is specialized for paths. DS5 is specialized for the representation of IC packages.

CHAPTER 3

Program use

3.1 General

Program MDAT is written in University of Minnisota Pascal version 2.0 and run on a CDC CYBER 175 computer. Fig. 35 shows various program options. The program without comments or debugging code is approximately 3600 lines and occupies 163 PRUs. The program with full debugging code is approximately 5500 lines and occupies 454 PRUs. The binary object file of the no debugging version is 260 PRUs.

PROGRAM	DECKS?	DPs?	CALLS?
MDATXXX	NO	NO	NO
MDATXDX	NO	YES	NO
MDATXDC	NO	YES	YES
MDATDDX	YES	YES	NO
MDATDDC	YES	YES	YES

DP = debugging procedures.

fig. 35: MDAT program options

3.2 Running the program

Assume the following file names and definitions:

LGO: Binary object file.

INPUT: File containing all non-catalog input information such as name-date-time tags, maximum running time (MAXTME), targets, weights, and tree-list. Fig. 36 gives a BNF description of this information (see 1.2.2 for detailed description).

OUTPUT: Contents described in 3.4 (see fig. 37).

CATIN: File containing catalog information including component type list and indexed component list as described in 1.2.1. Fig. 38 gives a BNF description of this information.

DATOUT: Contents described in 3.4. A BNF description is given in fig. 39.

To run MDATXXX execute the following control statements in either batch or timesharing mode:

```
RFL,50000.
SETTL, ttt.  (ttt is running time in octal ms.).
REDUCE(-).
LGO,XINPUT,XOUT,CATIN,DATOUT
```

```
{file}  ::=  {name-date-time}  {maxtime} {targets} {weights}
     {tree-list}
{name-date-time} ::= [3 alfa values] {eol}
{maxtime} ::= [+integer] {eol}
{targets} ::= {tarcost} {tarpow} {tarpacks} {eol}
{tarpow} ::= [+integer]
{tarpacks} ::= [+integer]
{weights} ::= {under-weights} {over-weights}
{under-weights} ::= {weight-list} {eol}
{weight-list} ::= {wtcost} {wtpow} {wtpacks}
{wtcost} ::= [+integer]
{wtpow} ::= [+integer]
{wtpacks} ::= [+integer]
{over-weights} ::= {weight-list} {eol}
{tree-list} ::=({tree}) {eolist-code}
{tree} ::= {delay} {tree-input-list}
{delay} ::= [+integer] {eol}
{tree-input-list} ::= ({tree-input}) {eolist-code}
{tree-input} ::= {number-of-inputs-this-level} {input-index}
     {delay-of-input} {eol}
{number-of-inputs-this-level} ::= [+integer]
{input-index} ::= [+integer]
{delay-of-input} ::= [+integer]
{eol} ::= [end-of-line-marker]
{eos} ::= [end-of-segment-marker]
{eolist-code} ::= -3 {eol}
```

fig. 36: BNF for file INPUT

```
GENMUX 00000000010000000001
CLOCK AT END READINFO = 14
CLOCK AT END GENCOL = 15
CLOCK AT END GENSET = 17
CLOCK AT END INTERSECT = 50
CLOCK AT END BRKUP = 56
CLOCK AT END MAKETREE = 60
CLOCK AT END PACKS = 65
CLOCK AT END RELATE = 66
CLOCK AT END PATH SELECT = 74
(line A) MAXIMUM TIME ALLOCATED = 7000,MAXIMUM  WORKOUT
TIME FOR
     OPTIMIZATION IS 4867
(line B) NEW MERIT FACTOR MAXIMUM AT CLOCK 91
(line  C)  NEW  MERIT  FACTOR  IS  6.1414473684210E-001
OPTIMUM COUNT IS 1
(line D) COMBINATIONS COMPLETED = 37 OUT OF 243
CLOCK AT END OPTIM = 93
NEW MERIT FACTOR MAXIMUM AT CLOCK 96
NEW  MERIT FACTOR IS 7.0701754385964E-001 OPTIMUM COUNT
IS 2
COMBINATIONS COMPLETED = 3 OUT OF 81
CLOCK AT END OPTIM = 242
(line  E)  NEW  BEST  CONFIGURATION WITH MERIT FACTOR =
2.1666666666667E-001
(line F) BEST COST = 960 BEST POWER = 1200
~^(line G) BEST PACKS = 8
CLOCK AT END GENBEST = 248
NEW MERIT FACTOR MAXIMUM AT CLOCK 255
NEW  MERIT FACTOR IS 6.1414473684210E-001 OPTIMUM COUNT
IS 83
COMBINATIONS COMPLETED = 13 OUT OF 27
CLOCK AT END OPTIM = 257
NEW MERIT FACTOR MAXIMUM AT CLOCK 260
NEW MERIT FACTOR IS 7.0701754385964E-001 OPTIMUM  COUNT
IS 84
                    .
                    .
CLOCK AT END OPTIM = 291
CLOCK AT END GENBEST = 294
CLOCK AT END PROGRAM = 307
```

fig. 37: Example of file OUTPUT

```
{file} ::= ({segment}) {eos}
{segment} ::= {mux-segment} | ....
{mux-segment} ::= {type-list}
{type-list} ::= ({type}) {eolist-code}
{type} ::= {number-of-components-in-package} {eol}
     {entry-list}
{number-of-components-in-package} ::= [+integer]
{entry-list} ::= ({entry}) {eolist-code}
{entry} ::= {cost} {delay} {power} {eol}
{cost} ::= [+integer]
{delay} ::= [+integer]
{power} ::= [+integer]
```

fig. 38: BNF for file CATIN

```
{file} ::= {name-date-time} {statistics} {tree-list}
{name-date-time} ::= [3 alfa values] {eol}
{statistics} ::= {cost} {power} {packs} {merit-factor}
{cost} ::= [+intger] {eol}
{power} ::= [+integer] {eol}
{packs} ::= [+integer] {eol}
{merit-factor} ::= [+integer] {eol}
{tree-list} ::= ({tree}) {eolist-code}
{tree} ::= {tree-index} {eol} {component-list}
{tree-index} ::= [+integer]
{component-list} ::= ({component}) {eolist-code}
{component} ::= {component-index} {catalog-info}
     {common-info} {eol} {input-info}
{component-index} ::= [+integer]
{catalog-info} ::= {catalog-type} {catalog-index}
{catalog-type} ::= [+integer]
{catalog-index} ::= [+integer]
{common-info} ::= {tree-index} {component-index}
{input-info} ::= {input-list} {internal-connects-list}
{input-list} ::= ({input} {eol}) {eolist-code}
{internal-connects-list} ::= ({component-index} {eol})
     {eolist-code}
{input} ::= {input-index}
{input-index} ::= [+integer]
{eol} ::= {end-of-line-marker}
{eolist-code ::= -3 {eol}
```

fig. 39: BNF for file DATOUT

(where XINPUT and XOUT are the same files as INPUT and OUTPUT but must be specified as XINPUT and XOUT so that they will not be interpreted as input from a terminal and output to a terminal). If XINPUT and XOUT are omitted as in

```
LGO,,CATIN,DATOUT.
```

then the input information is taken from a terminal in timesharing and what would normally go into file XOUT will be printed at a terminal. To run all versions of MDAT other than MDATXXX the following files must be used:

```
LGO,XINPUT,XOUT,XDEBUG,CATIN,DATOUT.
```

File XDEBUG will be discribed in 3.3.

3.3 Debugging

To facilitate future changes in the program, many versions of the program include procedures for debugging. If an entry in fig. 35 has the entry "calls" it means that the calls to debugging procedures were left in to give full debugging capability. If selective debugging is desired, use a version with "no calls" and put in call statements

where desired. The parameters necessary for calls to debugging procedures should be obvious by looking at the code for a particular procedure.

When using a version of the MDAT which has "debugging procedures," file XDEBUG must be used. File XDEBUG contains the entries for an array declared as:

```
CONTRL: ARRAY [1..MAXCNTRL] OF INTEGER.
```

The inputs are integers with values 0 or 1, ten integers per line with each line ending in a dummy character (see fig. 40 for an example). Each value in CONTRL controls the turning on or off (0 = off, 1 = on) of outputs to file OUTPUT as a result of calling a debugging procedure. Thus selective writing of output can be achieved making the writing of possibly useless output unnecessary. This feature can be very helpful if finely detailed debugging is necessary towards the end of a program. If the same level of detail were maintained throughout, huge output files would be generated. The correspondence between a CONTRL entry and the unit of debugging controlled may be gleaned from the appropriate version of the program since the numbering of debugging units is approximately sequential.

```
**************************************************************

                    0001010111 *
                    0101110101 *
                    1011010110 *
                    1011101011 *
```

fig. 40: Example of file XDEBUG

```
**************************************************************
```

Inspection of fig. 35 reveals "no decks" and "decks" entries. "No decks" refers to the standard form of files for use by text editors such as University of Illinois BOSS or ICE. "Decks" means that the file is in a special format for use by text editor BOS (available under UN = 3TWOSIX). Note that since BOS and BOSS are interpreted as being the same in timesharing mode, change to batch mode of a terminal before using BOS.

Decks are used as follows: the text of a deck is all the text between two *DECK lines in a source file. A *CALL in the source will substitute the text of the corresponding deck in place of *CALL (see fig. 41 for an example). A WD (write deck) command makes the substitution for *CALLs and removes *DECK marks.

By removing *CALLs for debugging procedures that are not desired, whole sections of code may be easily removed and a streamlined source file may be contructed with customized debugging.

3.4 Output description

The MDAT has two different output files, OUTPUT and DATOUT. OUTPUT is oriented towards stand alone operation of

****

Source file is as follows:

```
*DECK MAIN
     STATEMENT 1
     STATEMENT 2
*CALL EXPL1
*CALL EXPL2
     STATEMENT 3
*DECK EXPL1
     STATEMENT 4
     STATEMENT 5
*DECK EXPL2
     STATEMENT 6
     STATEMENT 7
```

If using test editor BOS and WD MAIN is executed, local file MAIN will contain

```
STATEMENT 1
STATEMENT 2
STATEMENT 4
STATEMENT 5
STATEMENT 6
STATEMENT 7
STATEMENT 3
```

fig. 41: Use of decks and calls

****

the MDAT and detailed information on the runtime operation of the program; wheras DATOUT is oriented towards MDAT use as a component program in an overall design automation system where detailed knowledge of the runtime workings of MDAT is not needed.

The information in the first part of OUTPUT is unique to this file. The clock time in milliseconds at the ends of key procedures are shown as "CLOCK AT END XXXXX" in fig. 37. "WORKOUT TIME FOR OPTIMIZATION" (line a) is the amount of time the program allocated to procedure OPTIM out of the total time available (MAXTME). If a new merit factor maximum is found in procedure GENOPTIM, the clock and new merit factor are printed (lines b and c). Also under "COMBINATIONS COMPLETED", the total number of combinations of indexed components out of the total number of possible combinations is written. The number of combinations may be one more than actually completed due to some peculiarities of the program. After each IAB, if there is a new best configuration, the new merit factor with package constraint is printed along with the BESTCOST, BESTPOWER, and BESTPACKS (lines e, f, and g). Note that as levels are fixed the total possible number of configurations decreases dramatically. If there is program termination due to exceeding time constraints in OPTIM, a message will be written indicating the last tree level fixed, the clock at

termination, and the number of combinations completed at termination.

The second part of OUTPUT and DATOUT convey basically the same information in different formats. Referring to fig. 39, "statistics" gives BESTCOST, BESTPOWER, and BESTPACKS and the corresponding merit factor as determined by GENBEST. "Tree" is a tree of a MOMUX. "Tree-index" is a numbering assigned to these trees which corresponds to the order in which they were read in by procedure READINFO. The "component-list" is a list of CTs for a particular tree. "Component-index" is a unique numbering given to each CT regardless of tree affiliation. "Catalog-info" gives indexed component information which was determined in OPTIM, GENOPTIM, and GENBEST. If the CT is part of a VCC then "common-info" gives the tree number and "component-index" of the lowest numbered tree which has a CT which is part of the VCC. The "component-index" of this CT is the "component-index" in "common-info". If no VCC is involved, both components of "common-info" are set to zero; this is the only case yielding a zero for these two components. "Input-info" has two components, "input-list" and "internal-connects-list"; one but not both of these is null (no data, just "eolist-code"). If the CT of this "component" is not a leaf of the MOMUX tree, no outside inputs connect to this CT, and "internal-connects-list"

lists the "component-index"s which are the roots of subtrees connected to this CT. If the CT of this "component" is a leaf, then outside inputs connect to this CT, and "input-list" lists the inputs which connect to this CT.

The corresponding information in file OUTPUT is presented in tree form. The nodes of the trees correspond to the INDEX1 field of TREENODE records in the DS3 data structure and are unique for each tree. Placement of components is indicated by covered nodes in the tree rather than by specifying the "input-list" or "internal-connects-list". Commonality is indicated by a statement such as "COMMON WITH COMP 1 OF TREE 1" and VCCs are dealt with as described above. The TYPE and INDEX entries are identical with "catalog-info". In addition the cost, power, and delay associated with the catalog entry is given as COST, POWER, and DELAY. "Input-info" is given as INPUTS=. The delays read in by READINFO are given by DELAY= immediately after INPUTS=. The trees are ordered by "tree-index".

3.5 Running time considerations

In order to guarantee termination of the MDAT in the time allocated (MAXTME), procedure FINDTIMES allocates time

to the various procedures of the MDAT. Most of the CPU time used in normal application of the MDAT will be in procedure OPTIM. Constant times large enough for most input problems are allocated for all other procedures and the remainder goes to OPTIM and GENOPTIM. To fine tune the allocation of time in OPTIM, the number of indexed component combinations is calculated for each IAB and for each fixed tree level state (see 2.9 for explanation of level fixing) and assigned to variable SHARE. If SHARE is divided by the total number of combinations for all IABs and all levels, amd this fraction multiplied by the total time assigned to OPTIM, then each call to the OPTIM/GENOPTIM pair may be given a maximum running time, variable TIME referenced by IAB number and fixed tree level state. If this time is exceeded, the best result up to that time is taken as the input to GENBEST and a message is written to file OUTPUT as indicated in 3.4.

The program will not enter OPTIM if the time allocated for OPTIM and GENOPTIM is determined to be insufficient for the size of the input problem. The minimum WRKTME allowed is LEVCNT * M * MINOPTTME where LEVCNT is the maximum number of levels in any tree in the MOMUX, M is the number of trees, and MINOPTTME is a constant presently set at 50 ms. If the time allocated is below the minimum, an error message "INSUFFICIENT TIME TO OPTIMIZE - INCREASE TIME" will be written to file OUTPUT.

CHAPTER 4

Results

4.1 Table and description of results

The results of running the MDAT on various problems is shown in fig. 42. Some entries have been omitted since they either are not normally reported in the output and must be determined by hand (PATHS) or the problem was computed with earlier versions of the MDAT which don't have the reporting facilities of the newest version (COMB).

****

PROBLEM	TT	TI	VCCs	PATHS	COMB	RT	AC
1	3	32	3	13	15750	12.5	100%
2	4	26	2	13	-----	6.5	100%
3	4	32	2	10	12601	7.7	100%
4	2	24	2	9	406880	97.7	100%
5	3	31	2	12	3081	2.0	100%
6	3	18	2	8	-----	5.9	100%
7	3	48	3	--	1065220	77.1	100%
8	3	48	3	14	16656	8.2	100%
9	4	32	4	11	391250	317.0	33%
10	4	24	4	11	-----	2.2	100%
11	4	24	4	--	-----	1.8	100%
12	4	34	5	13	10156875	317.0	6%
13	4	28	3	5	-----	0.2	100%
14	1	8	0	1	-----	0.7	100%
15	3	12	1	1	-----	0.1	100%

TT = total number of trees in MOMUX.
TI = total number of inputs to MOMUX.
COMB = total number of combinations of indexed components which MDAT must look at.
RT = runtime
AC = average % of combinations completed.

fig. 42: Results

****

REFERENCES

[1] W. M. van Cleemput and E. A. Slutz, "Initial Design Considerations for a Hierarchical IC Design System," Asilomar Conference on Circuits and Systems, Pacific Grove, CA, pp. 334-341, Nov. 1977.

[2] D. G. Ressler, "A Simple Computer-Aided Artwork System that Works," Proceedings of the 11th Design Automation Workshop, Denver, CO, pp. 92-97, June 1974.

[3] K. R. Stevens, W. M. van Cleemput, T. C. Bennett and J. A. Hupp, "Implementation of an Interactive Printed Circuit Design System," Proceedings of the 15th Design Automation Conference, pp. 74-81, 1978.

[4] D. L. Dietmeyer, "Introducing DDL," IEEE Computer, pp. 34-38, Dec. 1974.

[5] F. J. Hill, "Introducing AHPL," IEEE Computer, pp. 28-30, Dec. 1974.

[6] C. G. Bell and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, 1971. (ISP language.)

[7] M. B. Baray and S. Y. H. Su, "A Digital System Modeling and Design Language," Proceedings of the 8th Design Automation Workshop, pp. 1-22, 1971.

[8] W. M. van Cleemput, "An Hierarchical Language for the Structural Description of Digital Systems," Proceedings of the 14th Design Automation Conference, New Orleans, pp. 377-385, June 1977.

[9] C. W. Rose and M. Albarran, "Modeling and Design Description of Hierarchical Hardware/Software Systems," Proceedings of the

12th Design Automation Conference, Boston, MA, pp. 421-430, June 1975.

[10] U. R. Kodres, "Partitioning and Card Selection," Ch. 4 of Design Automation of Digital Systems, ed. by M. A. Breuer, Prentice Hall, 1972.

[11] D. C. Schmidt, "Gate for Gate Modular Replacement of Combinational Switching Networks," Proceedings of the 9th Design Automation Workshop, Dallas, TX, pp. 331-340, June 1972.

[12] D. C. Schmidt and G. Metze, "Modular Replacement of Combinational Switching Networks," IEEE Transactions on Computers, Vol. C-24, No. 1, pp. 29-46, Jan. 1975.

[13] L. J. Hafer and A. C. Parker, "Register-Transfer Level Digital Design Automation: The Allocation Process," Proceedings of the 15th Design Automation Conference, pp. 213-219, 1978.

[14] E. J. Snow, D. P. Siewiorek and D. E. Thomas, "A Technology-Relative Computer-Aided Design System: Abstract Representations, Transformations, and Design Tradeoffs," Proceedings of the 15th Design Automation Conference, pp. 220-226, 1978.

[15] R. J. Kinch, "Computer-Aided Design of Computer Architectures Using Interactive Graphic Representation," M.S. thesis, University of Illinois at Urbana-Champaign, Department of Computer Science Report No. 78-952, Dec. 1978.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-79-972	2.	3. Recipient's Accession No.

4. Title and Subtitle

Automated Design of Digital Multiplexers

5. Report Date: June, 1979

6.

7. Author(s): Albert Ernest Casavant

8. Performing Organization Rept. No. UIUCDCS-R-79-972

9. Performing Organization Name and Address

University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.: US NSF MCS77-27910

12. Sponsoring Organization Name and Address

National Science Foundation
Washington, D. C.

13. Type of Report & Period Covered: Master's Thesis

14.

15. Supplementary Notes

16. Abstracts

This paper describes the automated design of digital multiplexers. Using a computer software tool, simple non-technical descriptions of multiplexer tree(s) may be translated into a hardware implementation using building block components from a predefined library of components. The program handles commonality among trees and variations in the delay of library components. The user has direct control over the weights given to various design criteria as well as the tradeoff between the quality of the design and the amount of computer time consumed in the automated design process. Also described in detail are the assumptions made and the heuristics employed in the program, the functions of critical parts of the code, and a user's guide. The program is written in PASCAL 6000 version 2 and runs on a CYBER 175.

17. Key Words and Document Analysis. 17a. Descriptors

Automated logic design
Computer-aided design
Design automation
Logic design
Multiplexers

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement: Release Unlimited

19. Security Class (This Report): UNCLASSIFIED

20. Security Class (This Page): UNCLASSIFIED

21. No. of Pages: 104

22. Price